UNIVERSITÀ DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

Progettazione e sviluppo di una infrastruttura hardware e software per un sistema di supporto integrato in ambiente domestico

Primo Relatore: Luca Fanucci
Secondo Relatore: Pierfrancesco Foglia
Terzo Relatore: Simone Perini

Candidato: Francesco Sechi

Anno Accademico 2006/2007

Ringraziamenti

Ringrazio i miei genitori per aver creduto in me e avermi sostenuto nel lungo percorso di formazione, di cui questa attività è conclusione.

Ringrazio tutta la squadra dell'Amic S.r.l. che mi ha dato la possibilità di mettermi alla prova, offrendomi tutta la loro disponibilità e pazienza.

Ringrazio il prof. Luca Fanucci per avermi assistito durante l'intera attività di tesi.

Ringrazio infine tutte le persone che mi son state vicine in questi anni.

Sommario

Indice delle figure

Indice delle tabelle

Prefazione

L'attività di tesi tratta lo sviluppo di uno dei moduli costituenti il progetto ELIK dell'Amic S.r.l..

Amic S.r.l. è un'azienda livornese appartenente al gruppo Microtel, costituito da diverse società europee e operante nel settore dell'*Electronic Manufacturing Services* (EMS). Essa è una delle aziende ufficialmente riconosciute come *Spin-Off* della Scuola Superiore Sant'Anna di Pisa e la sua missione è quella di realizzare dispositivi e sistemi elettronici partendo dalle specifiche del cliente, attraverso accurate fasi di progettazione, scelta materiali, sviluppo e prototipazione.

Il progetto ELIK (acronimo di Easy Living in Kitchen) si colloca nella disciplina dell'ICT nota come domotica, che comprende l'insieme dei concetti applicati all'ambiente domestico per l'integrazione delle tecnologie facenti parte dell'impiantistica tradizionale con quelle di nuova generazione, al fine di ottenere nuove e più complesse funzionalità. Le tecnologie adottate in domotica sono diverse e nessuna può essere considerata uno standard *de facto*: questo è uno dei fattori che contribuisce al rallentamento della crescita del mercato dell'*home automation*, anche se in questi ultimi anni quest'ambito è in notevole espansione. Le aziende che stanno cercando di imporsi sul mercato domotico sono parecchie, e quindi vengono proposte soluzioni eterogenee: nessuna di esse, però, tiene in considerazione le peculiarità dei singoli ambienti che compongono l'abitazione.

Il progetto tema della tesi si pone come primo obiettivo quello di dare la giusta specializzazione al vano cucina, dato che questo è l'ambiente che più presenta caratteristiche singolari rispetto agli altri vani. Questo risultato si ottiene dotando la cucina di un sistema intelligente che sia in grado, sfruttando la conoscenza sugli utenti acquisita dai *sistemi esperti* di cui è provvisto e le potenzialità degli elettrodomestici, di

assistere le attività dell'utente e di garantirgli un elevato grado di sicurezza, segnalando eventuali anomalie e malfunzionamenti e attuando misure protettive.

Inoltre ELIK, data la sua modularità e scalabilità, può essere pensato come base tecnologica sulla quale impostare future applicazioni, come ad esempio lo *smart remote shopping*.

L'attività di tesi consiste nello sviluppo dell'hardware e del firmware di una delle componenti che costituisce il sistema complessivo: la scheda di controllo. Essa, sfruttando le potenzialità dei microcontrollori della famiglia ARM7, permetterà di dotare gli elettrodomestici di un grado di intelligenza tale da consentire loro di comunicare e interagire con la scheda master, sulla quale verranno installate le applicazioni ad alto livello che gestiranno il *sistema esperto*.

Il progetto si può suddividere in diverse fasi, ognuna delle quali facilmente inquadrabile nel piano di lavoro dell'azienda.

La fase iniziale sarà di acquisizione, da parte del candidato, del know–how già appreso dall'azienda nell'ambito della programmazione del microcontrollore LPC2148 della Philips e di apprendimento delle specifiche del sistema.

La seconda sarà di comprensione delle specifiche e di sviluppo del protocollo ModBus. A questa fase seguirà un'attività di ricerca di un ambiente di sviluppo che offra il giusto trade–off fra efficacia e costo; il protocollo ModBus sarà quindi "portato" su tre compilatori diversi, in modo da testare le potenzialità dei singoli ambienti di sviluppo.

Successivamente, si passerà allo sviluppo su microcontrollore ARM7 LPC2378 della Philips, dotato di controller Ethernet e USB, per il quale, inizialmente, si implementerà la parte della libreria firmware utile allo sviluppo del sistema ELIK.

In seguito, su questa scheda, si eseguirà il porting del protocollo opensource *TCP/IP LightWeight IP* sviluppato da Adam Dunkels del

SICS, sul quale si implementerà poi il protocollo proprietario *AmicLan* dell'Amic S.r.l..

Al termine di questa fase si svilupperà il protocollo AmicLan over USB previa analisi del protocollo USB, durante la quale si è prestata particolare attenzione allo studio delle classi CDC per la scelta del tipo di comunicazione ideale.

Per concludere, si effettuerà il porting del protocollo ModBus sul microcontrollore LPC2378, così da permettere l'integrazione dei vari moduli.

1. Introduzione

1.1 La domotica

Sin dai tempi degli antichi romani la casa non era solo un ambiente dove assolvere alle necessità fisiche fondamentali, ma anche un sito che offriva servizi e spazi destinati allo sviluppo della vita culturale e sociale della famiglia. Parallelamente all'evolversi della società e della tecnologia sono mutate anche le esigenze delle persone che vivono e abitano l'immobile. I requisiti di un ambiente domestico, in termini di capacità di soddisfare le necessità dei suoi occupanti, non sono più quindi solamente legati ad aspetti strutturali come spessore dei muri, grado di illuminazione e così via, ma anche alla capacità di offrire servizi e comfort e quindi alla sua dotazione impiantistica. Inoltre si avverte sempre più la necessità che questi servizi, siano essi di base o evoluti, siano facilmente accessibili e semplici da gestire, minimizzino i costi e offrano continuità di esercizio e scalabilità.

Tutto ciò ha portato ad un utilizzo sempre più intensivo dei sistemi automatizzati nelle abitazioni, affiancando così ai dispositivi tradizionali delle componenti elettroniche che permettessero di ampliare le loro capacità operative. E' cambiato così il modo di vedere l'ambiente domestico, non più come un sistema di oggetti che si acquistano nel tempo, ma come sistema di servizi offerti attraverso terminali che dovranno essere sempre più interagenti fra loro.

Si è assistito così alla nascita della disciplina della *Domotica* (nota anche come *Home Automation*), che si pone l'obiettivo di integrare le tecnologie dell'impiantistica tradizionale già presenti negli edifici con quelle innovative, al fine di ottenere nuove e moderne funzionalità. Con integrazione si intende, al riguardo, la condivisione dell'informazione fra i singoli impianti installati, che possono così cooperare per realizzare nuove funzioni complesse: questo significa

potenziare le capacità del sistema complessivo pur conservando le specializzazioni e competenze dei diversi operatori. Con l'integrazione si ha quindi la scissione della parte di attuazione dalla parte di informazione, mettendo quest'ultima a disposizione dell'intero edificio.

In un sistema di home automation le informazioni vengono raccolte dal campo da sensori, che permettono di catturare eventi (per esempio fughe di gas), informazioni (per esempio valori di temperatura) o volontà (per esempio l'azionamento di interruttori della luce). Gli attuatori invece hanno il compito di svolgere un'azione, generalmente in seguito al verificarsi di un evento registrato da un sensore. Questa separazione porta l'impiantistica domotica ad un'evoluzione rispetto agli impianti elettrici tradizionali: si passa dal circuito fisico, cioè l'insieme delle linee di potenza che collegano i sensori di evento (ad esempio un pulsante) al carico, al circuito logico che trasporta informazioni dai sensori agli attuatori; aumentano così la modularità, la scalabilità e la modificabilità, dato che qualsiasi variazione del sistema implica una modifica nella gestione delle informazioni, e non nel cablaggio dello stesso circuito fisico.

Per il trasporto dell'informazione si rende a questo punto necessario definire un protocollo di comunicazione e un mezzo trasmissivo: esistono attualmente numerosi standard utilizzati nel campo dell'automazione domestica, e uno dei fattori che ha contribuito a rallentare la crescita di questo settore è proprio l'incertezza su quale di essi prevarrà e diventerà lo standard di mercato.

I protocolli utilizzati in campo domotico si possono classificare in due macrocategorie: quelli di tipo *proprietario* e quelli *aperti*.

I protocolli *proprietari* sono quelli realizzati da un singolo costruttore che non ha interesse a divulgare le specifiche del prodotto e ne rende impossibile l'utilizzo libero a terzi.

I protocolli *aperti*, al contrario, sono quelli per i quali vengono divulgate le specifiche e che quindi possono essere utilizzati liberamente da qualsiasi azienda per sviluppare i propri prodotti (e per questo vengono anche definiti standard).
I principali protocolli utilizzati nell'ambito della domotica sono:

- IrDA: protocollo per la connessione a raggi infrarossi, il cui limite è il raggio di funzionamento e la necessità di visibilità ottica;
- Bluetooth: protocollo per la connessione senza fili di dispositivi elettronici diversi;
- UPnP: protocollo basato sui protocolli Web standard;
- Jini: protocollo presentato come standard industriale aperto, che propone un modello di networking astratto che vede ogni elemento della rete come oggetto che offre dei servizi;
- Protocolli industriali: ModBus, ProfiBus, FieldBus, e così via.

Le informazioni possono viaggiare su cavo o via etere. Fra i mezzi trasmissivi del primo tipo rientrano il doppino (twisted pair), il cavo coassiale, la fibra ottica o cavi multipolari specifici; via etere, invece, si può trasmettere in radiofrequenza o con raggi infrarossi. Nonostante la loro limitatezza in banda non va trascurata la possibilità di utilizzare linee di potenza come mezzo trasmissivo, dal momento che sono sicuramente presenti nell'ambiente domestico.
I dispositivi che permettono la realizzazione di impianti domotici possono essere classificati in base alla loro funzionalità come:

- Dispositivi di comando e sensori: hanno il compito di rilevare e misurare le grandezze fisiche di interesse nell'ambiente in cui si trovano; possono essere dummy, quindi hanno bisogno di intelligenza aggiuntiva per entrare a far parte di un impianto domotico, o smart, quindi già dotati di interfacce per colloquiare all'interno della rete;

- Dispositivi di uscita: ad essi sono collegati direttamente o indirettamente i carichi elettrici;
- Sistemi di controllo e gestione: introducono l'intelligenza nella rete domotica;
- Dispositivi di sistema: rientrano in questa categoria gli alimentatori, i dispositivi di accoppiamento e l'interfaccia del sistema domotico con il PC;
- Interfacce e residential gateway: dal punto di vista dell'utente è l'elemento più importante, perché è quello che gli permette di interagire con il sistema. Esso deve essere perciò funzionale e facile da utilizzare.

L'integrazione delle componenti elettroniche costituenti un ambiente domestico porta numerosi vantaggi sotto diversi punti di vista: oltre a rendere più funzionali ed efficienti i singoli componenti, sfruttando la loro integrazione per ottenere nuove funzioni complesse, porta benefici anche dal punto di vista socio-ambientale.
Fra i più rilevanti è sicuramente da citare il risparmio energetico: infatti un sistema domotico, abbinato ad un sistema di controllo dei consumi, può contribuire ad equilibrarli, attenuando i picchi ed eliminando gli sprechi; inoltre, la domotica permette di sfruttare al meglio le risorse ottenibili dalle fonti rinnovabili, riducendo così l'impatto ambientale.
Un altro aspetto positivo è il miglioramento della qualità della vita di quelle persone che, per deficit fisici o per cause sociali, si trovano in condizioni di svantaggio rispetto alla maggioranza della popolazione. Ad esempio, un'installazione domotica permette di aumentare l'autonomia delle persone con deficit motori sia riducendo lo sforzo fisico, sia facilitando l'accesso ai mezzi di comunicazione.
Un'ulteriore esigenza fortemente sentita è la sicurezza degli edifici, intesa sia in termini di security che di safety: un impianto domotico

permette di garantire entrambe, tramite sistemi di antifurto e controllo degli accessi per quanto riguarda il primo aspetto, con impianti antincendio, rilevatori di fughe di gas, e così via, per quanto riguarda il secondo.

1.2 Stato dell'arte

Nonostante la domotica sia ancora un settore in via di sviluppo, sul mercato esistono diverse soluzioni per l'home automation. Ognuna di esse offre la possibilità di controllare tutti i servizi elettrici connessi al sistema e di gestire l'impianto di anti-intrusione e videosorveglianza. Generalmente sono dotate di interfaccia a menù offerta su display LCD o touchscreen e molte permettono anche il controllo da remoto tramite internet, telefono cellulare o linea telefonica tradizionale. Per esempio, i sistemi *MyHome* della BTicino e *Hoasis* della BPT permettono di accedere ai servizi offerti dal sistema attraverso un portale accessibile via internet; *ByMe* di Vimar e *Sistema Casa 2000* di Sistema Casa permettono il controllo dell'impianto domotico attraverso un qualsiasi comunicatore telefonico; ancora, *Chorus* della Gewiss e *Semplice* di Urmet consentono la ricezione di segnalazioni d'allarme e l'invio di comandi tramite SMS.

Sebbene i sistemi che utilizzano cavi per l'interconnessione dei vari dispositivi siano più diffusi, esistono soluzioni, come *Atmosphera* di BM S.p.A., che utilizzano connessioni wireless: esse hanno come punto di forza la non invasività, che permette l'installazione dell'impianto senza lavori edili supplementari.

Solitamente le soluzioni in commercio prediligono l'utilizzo di protocolli aperti: ad esempio, *Domino* di Duemmegi utilizza un protocollo simile al ModBus, mentre i già citati *Sistema Casa 2000*, *Atmosphera* e *ByMe* utilizzano rispettivamente X10, ZigBee e Konnex.

Esistono comunque diversi impianti che utilizzano soluzioni proprietarie: per esempio, il sistema *MyHome* della BTicino utilizza un bus proprietario basato su due cavi twistati, utilizzati sia per il trasporto di comandi digitali sia per l'alimentazione. Altri esempi sono i sistemi della Crestron che utilizzano il protocollo di rete Cresnet, quelli della Vantage che utilizzano un protocollo proprietario di tipo Plug&Play e *PICnet* dell'italiana Sinthesi che utilizza una coppia di cavi twistati per la trasmissione di dati e una coppia per l'alimentazione.

Dato che le soluzioni di tipo proprietario potrebbero rendere il sistema "chiuso", quindi non integrabile con soluzioni di terze parti, di solito le aziende che adottano questa scelta dotano i propri sistemi di interfacce che permettono la comunicazione verso altri tipi di bus.

Il quadro del mercato della domotica appena delineato mostra come vi sia ancora una forte eterogeneità nelle soluzioni, dal punto di vista dei protocolli e dei mezzi di comunicazione utilizzati. Tuttora non si è affermato uno standard *de facto* e questo, assieme alla diffusione di protocolli proprietari, rende difficile l'integrazione della domotica in apparati come i residential gateway e i set-top box, ostacolando quindi la naturale evoluzione della casa verso il concetto di ambiente intelligente.

1.3 La domotica in Italia

Uno dei problemi principali nella valutazione dell'andamento del mercato della domotica è dovuto all'identificazione di quali installazioni possano essere considerate appartenenti a questa categoria.

Sebbene da un'analisi di mercato risulta che, complessivamente, il mercato europeo della domotica sia fortemente in crescita, risulta una forte divergenza fra l'Europa del nord, dove le imprese del settore

sono molto attive, e quella dell'area mediterranea, dove il mercato va ancora a rilento, anche se vi sono aspettative di crescita per gli anni a venire.

In Italia, in particolare, il mercato della domotica presenta un importante trend evolutivo che ha portato, negli ultimi anni, ad una crescita degli investimenti nel settore. Nonostante ciò, sebbene l'interesse da parte dell'utente finale sia in crescita, sono ancora poche le aziende in grado di offrire soluzioni integrate e integrabili, per via delle loro ridotte dimensioni. Pur con le difficoltà citate vi è consapevolezza da parte delle imprese che questo è un momento favorevole per operare verso l'innovazione delle funzionalità di base.

Da uno studio sull'evoluzione dell'home automation nel mercato italiano a proposito dei sistemi di sicurezza e teleservizi, effettuato nel 2003 da Sistema Casa, risulta che, sebbene queste due aree rappresentino ancora l'80% degli investimenti effettuati in ambito domotico, nel prossimo futuro una percentuale sempre maggiore del mercato complessivo sarà destinata al comparto dell'home automation.

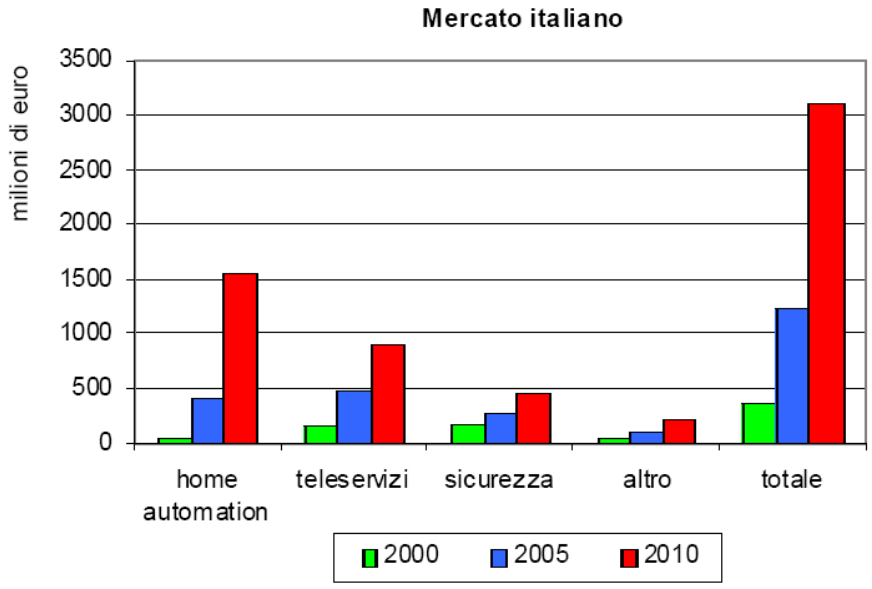

Figura 1: Evoluzione dell'home automation in Italia (fonte: Sistema Casa)

2. Il progetto

2.1 Introduzione

Dall'analisi di mercato si è constatato come i sistemi esistenti offrano soluzioni, in alcuni casi piuttosto evolute, per l'automatizzazione della casa e il controllo completo e l'interazione dei dispositivi elettronici installati nell'ambiente domestico. Tutte permettono la gestione dei carichi e il monitoraggio dello stato di sicurezza della casa, oltre al controllo di situazioni anomale e malfunzionamenti. Sebbene tutte raggiungano l'obiettivo prepostosi dall'home automation, seppur con un diverso grado di complessità, esse trattano l'ambiente domestico nella sua interezza, senza tener conto delle eterogeneità fra i singoli ambienti che lo compongono. Infatti, essendo pensate per la gestione globale della casa, esse permettono di controllare le funzionalità che accomunano tutti i vani, senza però alcun grado di specializzazione.

La cucina, invece, presenta caratteristiche del tutto singolari rispetto agli altri vani della casa: essa, infatti, riflette più di ogni altro la personalità degli abitanti della casa, e per questo motivo non può essere trattata al pari degli altri ambienti.

Il progetto ELIK si propone l'obiettivo di colmare questa lacuna nel mercato con lo sviluppo di un sistema che permette il controllo dell'ambiente cucina in tutta la sua complessità e il suo interfacciamento con la maggior parte delle soluzioni già presenti in commercio: esso sarà così in grado di offrire un servizio innovativo e all'avanguardia nel campo della domotica, sfruttando le potenzialità degli elettrodomestici e le molteplici fonti di informazione a cui sarà in grado di accedere (dieta, cartella clinica, e così via).

Inoltre ELIK, data la sua modularità e scalabilità, rappresenta una importante base sulla quale è possibile impostare tutta una serie di future applicazioni che coinvolgano anche altri soggetti commerciali.

Si può citare, a riguardo, lo "smart remote shopping", che offrirebbe la possibilità di stilare automaticamente liste della spesa in funzione di un insieme di dati a disposizione del sistema (prodotti presenti in dispensa, prodotti più utilizzati dalla famiglia in base agli acquisti pregressi, eccetera) o di far proporre al cliente, da parte del supermercato, le migliori offerte personalizzate del giorno; ancora, potrebbe permettere di effettuare ordini da remoto e consegne della spesa a domicilio, o il ritiro della stessa attraverso canali preferenziali messi a disposizione dal venditore.

2.2 ELIK: Easy Living in Kitchen

Il progetto ELIK prevede lo sviluppo di un sistema integrato in grado di rendere sicuro, intelligente e maggiormente usabile il sistema *cucina domestica*.

A tal proposito, si propone qui un sistema in grado di integrare una nuova generazione di elettrodomestici dotati di intelligenza superiore rispetto agli standard attuali, che ne permetta un utilizzo efficiente e sicuro: esso sarà in grado di svolgere attività complesse, sfruttando la capacità di comunicazione di cui gli stessi saranno dotati per condividere informazioni. Grazie all'utilizzo di moduli dotati di *algoritmi esperti* il sistema, a partire da una base di conoscenza formata dalle informazioni della filiera dei prodotti alimentari presenti in dispensa, dalla storia clinica, dai gusti, dalla dieta degli utenti e così via, potrà assistere le attività dell'utente della cucina, ad esempio offrendo informazioni sulla ricetta ideale in base a parametri impostati dall'utente (ricetta economica, oppure che utilizzi i prodotti in scadenza o, ancora, priva di allergeni, secondo le esigenze dei commensali), oppure con l'impostazione automatica degli attuatori (temperatura e tipologia di cottura del forno, selezione del lavaggio più indicato per la lavatrice, ecc.). In più, ELIK aumenterà il livello di

sicurezza in cucina, segnalando all'utente la presenza di eventuali anomalie (fughe di gas o acqua, variazioni anomale della temperatura degli elettrodomestici) o malfunzionamenti (notifica del guasto e *logging* degli eventi con segnalazione remota al centro assistenza).
Uno schema logico per tale sistema potrebbe essere il seguente:

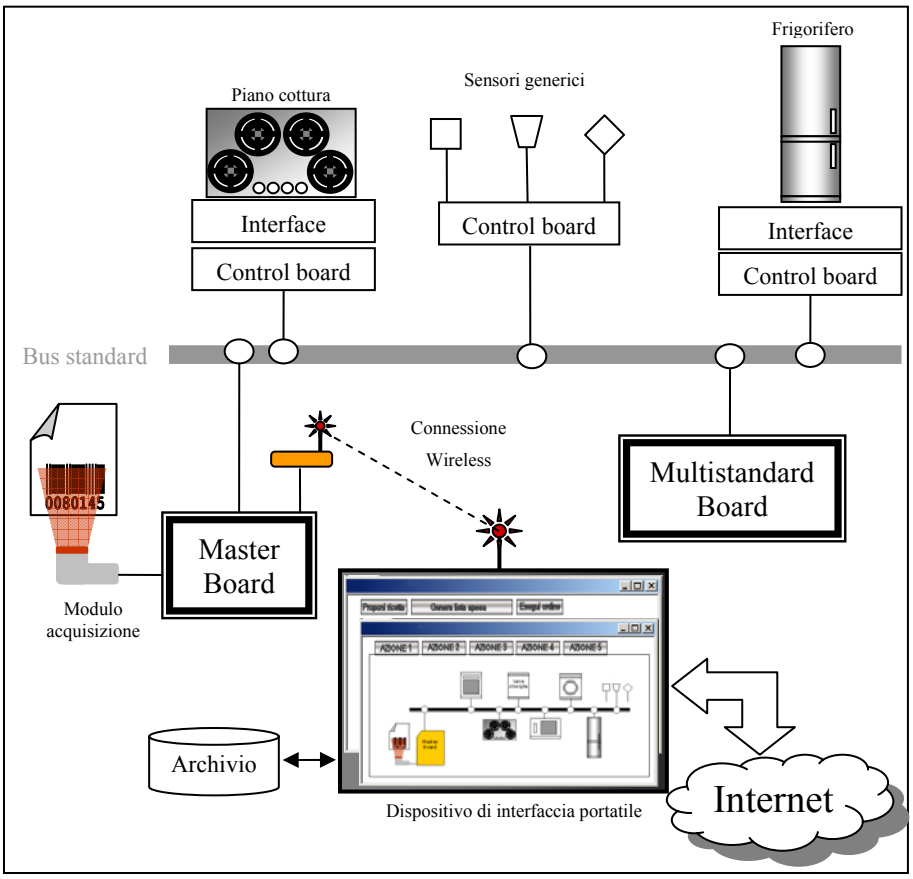

Figura 2: Esempio di architettura ELIK

Fra le finalità del progetto vanno sottolineati anche i vantaggi di tipo socio-ambientale che l'installazione di questo sistema integrato apporterebbe.
Un primo è una maggior tutela della salute dei lavoratori e una maggiore sicurezza dei luoghi di lavoro. Infatti, la cucina è il luogo di lavoro delle casalinghe e dei collaboratori domestici, e renderla intelligente e intrinsecamente più sicura (tramite sensorizzazione

dell'ambiente, segnalazione di possibili eventi dannosi) permette di raggiungere questo scopo.

ELIK, inoltre, rappresenta uno strumento di perseguimento delle pari opportunità, dove per pari opportunità si intende l'insieme degli interventi che a diversi livelli (comunitario, nazionale, regionale, locale), in diversi ambiti (politico-culturali, istituzionali, sociali, economici) e con diversi strumenti (legislativi, amministrativi, di iniziativa e di controllo), si propongono di riequilibrare le reali condizioni di svantaggio in cui, nella nostra società, si trovano tuttora a operare le donne rispetto agli uomini sotto il profilo delle opportunità loro offerte dall'attuale cultura ed organizzazione politica, civile e socio-economica. Lo studio e la realizzazione del sistema in oggetto, per le sue caratteristiche, incentiva il principio di pari opportunità poiché consente una migliore gestione dei lavori domestici, tuttora quasi esclusivo appannaggio del sesso femminile.

Ultimo, ma non meno importante, è il tema dell'impatto ambientale, che in ELIK si realizza nell'ottimizzazione dei consumi e nel conseguente risparmio energetico. Infatti, dotando di intelligenza i dispositivi che consumano la maggior quantità di energia in ambiente domestico, è possibile implementare politiche di gestione delle utenze in modo da ridurre l'impatto ambientale (per mezzo di segnalazione di inefficienze o scarsi rendimenti, scelta automatica di programmi di lavaggio *ad hoc*, minor consumo di agenti detergenti e di acqua, e via dicendo) e ottimizzare i consumi (ad esempio, distribuendo i carichi di lavoro in funzione delle fasce orarie con tariffe più basse).

2.3 Specifiche

Il progetto ELIK prevede lo sviluppo delle seguenti parti:

- una serie di moduli elettronici in grado di:
  - colloquiare tra loro;
  - comandare i vari attuatori degli elettrodomestici che compongono la cucina;
  - ricevere informazioni dai sensori presenti sugli elettrodomestici;
  - offrire un'interfaccia molto intuitiva e usabile per il controllo ed lo sfruttamento massimale del sistema da parte dell'utente;
  - offrire meccanismi intuitivi e rapidi di riconoscimento dei prodotti che vengono aggiunti alla dispensa;
  - offrire all'intero sistema un'interfaccia verso vari standard commerciali;
- una collezione di programmi per:
  - l'accesso a database condivisi per la memorizzazione delle informazioni riguardanti:
    - i possibili prodotti (alimentari e non) presenti nella cucina;
    - le persone che usufruiscono della cucina e/o che vengono invitate per i pasti;
  - l'implementazione dei comportamenti delle varie schede elettroniche;
  - l'interazione utente-sistema e quindi per:
    - l'impartizione dei comandi;
    - la visualizzazione dello stato dei vari dispositivi;
    - la proposta intelligente da parte del sistema di suggerimenti e/o la guida ragionata alla preparazione di piatti o alla spesa successiva che ottimizzi i parametri desiderati dall'utente (diete personali, prodotti in

scadenza prossima, prodotti provenienti da determinati luoghi, prodotti in offerta, ecc.);

- l'interrogazione del database dei prodotti per il loro corretto utilizzo e conservazione, la loro scadenza, ecc.;
- la segnalazione di eventuali anomalie presenti nel sistema cucina e la loro eventuale segnalazione a centri specializzati.

Queste specifiche verranno raggiunte implementando le seguenti unità (schede) elettroniche e il relativo firmware:

- **Control Board CB-1000:** controllo degli INPUT e degli OUTPUT per gli elettrodomestici, nonché dei sensori generici;
- **Control Board CB-2000:** controllo degli INPUT e degli OUTPUT per gli elettrodomestici (maggior numero di INPUT e di OUTPUT rispetto a CB-1000);
- **Interface Board UI-1000:** interfacciamento dell'elettrodomestico;
- **Interface Board UI-3000:** interfacciamento evoluto dell'elettrodomestico;
- **Master Board:** controllo del sistema cucina e della presentazione intelligente dei dati, e acquisizione della conoscenza (dotata di modulo wireless);
- **Multistandard Board:** apertura del sistema agli standard presenti sul mercato.

e i seguenti moduli software:

- modulo di interfaccia del sistema;
- modulo per l'accesso al database condiviso;
- modulo per l'interpretazione "intelligente" dei dati.

2.4 Piano di lavoro

In questo paragrafo vengono riportate, per completezza di esposizione, tutte le informazioni sul piano di lavoro del progetto ELIK, approfondendo le fasi di sviluppo riguardanti l'esperienza di tesi.

Il progetto è stato suddiviso in otto moduli, la maggior parte dei quali inerenti la progettazione e lo sviluppo di schede elettroniche e del relativo firmware, e quindi del sistema al più basso livello di sviluppo.

Si parte con un modulo di lavoro che prevede la raccolta sistematica delle specifiche, prestando particolare attenzione al mercato e allo stato dell'arte nel settore. Sebbene le specifiche siano state definite all'inizio dell'attività, grazie alle ridotte dimensioni del team di sviluppo e all'utilizzo di un approccio incrementale si sono potute apportare alcune modifiche in corso d'opera al fine di rendere il sistema più "aperto" e performante.

I successivi cinque work package riguardano la realizzazione di quattro schede elettroniche e di una collezione di librerie firmware che costituiranno l'infrastruttura hardware dell'architettura.

Ogni work package è ripartito in una serie di attività canoniche per il particolare sviluppo: si inizia con la progettazione hardware e firmware di ciascun modulo, si passa attraverso una prima fase di sviluppo, si crea un primo set di prototipi, si testano tali prototipi, si operano le eventuali modifiche HW e FW e, infine, si produce la release finale del modulo. Tali attività potranno prolungarsi oltre la durata prevista, in funzione degli aggiustamenti che si potrebbero rendere necessari al momento dell'assemblaggio dei singoli componenti nel dimostratore previsto dal progetto. Un modulo, che verrà attivato all'ottavo mese, riguarderà lo sviluppo del software che dovrà offrire all'utente gli strumenti per lo sfruttamento del sistema. Infine, l'ultimo modulo riguarda la costruzione di un dimostratore il cui scopo è evidenziare le potenzialità del sistema, i possibili miglioramenti e gli eventuali sviluppi futuri dello stesso.

Quindi, il piano di lavoro può essere schematizzato nel seguente modo:

Tabella 1: Piano di lavoro del progetto ELIK

Modulo di lavoroN.	Titolo del modulo di lavoro	Mesi/uomo	Mese di inizio	Mese di fine	N.Prodotto da fornire
WP 1	Definizione specifiche sistema	3,5	M1	M2	
WP 2	Progettazione e sviluppo protocollo Modbus	4,4	M2	M6	1
WP 3	Progettazione e sviluppo scheda multi-standard	4,7	M2	M7	2
WP 4	Progettazione e sviluppo schede controllo	6,2	M3	M18	3
WP 5	Progettazione e sviluppo scheda interfaccia	7,9	M3	M18	4
WP 6	Progettazione e sviluppo scheda gestione	10,3	M3	M18	5
WP 7	Progettazione e sviluppo SW	7,1	M8	M18	
WP 8	Progettazione e sviluppo dimostratore	2,3	M14	M18	
	TOTALE	46,4			

Per i vari moduli di lavoro elencati nella Tabella 1 è possibile illustrare le varie attività che lo compongono e i mesi di progetto impiegati per ognuna di queste mediante il **Diagramma di Gantt** mostrato in Figura 3:

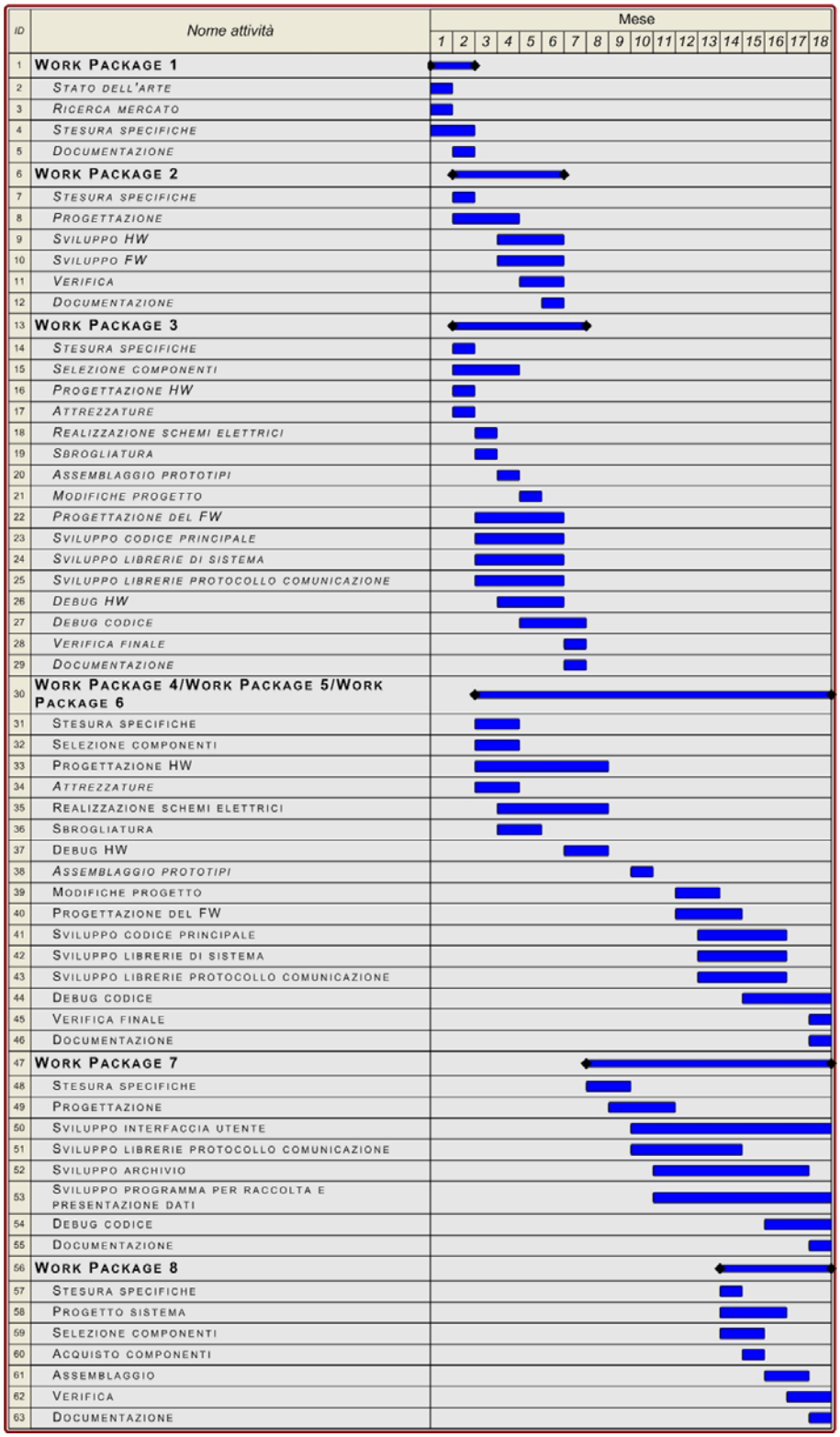

Figura 3: Diagramma di Gantt delle attività

I moduli concernenti il lavoro di tesi sono principalmente il WP2 e WP4, sebbene parte del lavoro svolto verrà utilizzato anche per la terminazione degli altri moduli.

2.5 Moduli di lavoro

Di seguito verranno descritti in dettaglio i moduli di lavoro caratterizzanti la fase di sviluppo del sistema ELIK.

- **WP1 - Definizione specifiche sistema:**
  L'obiettivo del primo modulo è quello di individuare dettagliatamente le specifiche del sistema in funzione dei prodotti già esistenti sul mercato e delle aspettative dello stesso, per mezzo di ricerca bibliografica sullo stato dell'arte del settore e dell'analisi delle esperienze degli operatori dei potenziali settori di interesse.

- **WP2 - Progettazione e sviluppo protocollo Modbus:**
  In questa fase vengono prodotte le librerie firmware per lo specifico microcontrollore che consentono l'interfacciamento del sistema con il protocollo di comunicazione Modbus, utilizzando la classica metodologia di produzione di librerie firmware che prevede varie fasi interattive di progettazione, sviluppo e debugging su sistema.

- **WP3 - Progettazione e sviluppo scheda multi-standard:**
  Questo modulo prevede la produzione di una scheda che consenta l'interfacciamento con più standard di comunicazione; in particolare, gli standard da interfacciare sono USB, RS485, RS232 ed Ethernet. La scheda da sviluppare potrà anche gestire un'uscita SPI ed I2C, e la

compatibilità con i suddetti protocolli sarà garantita almeno a livello PHY. Per essa è previsto anche l'isolamento galvanico.

- **WP4 - Progettazione e sviluppo scheda controllo:**
  Lo scopo di questa attività è la produzione di una scheda che consenta il controllo di un singolo elettrodomestico.

- **WP5 - Progettazione e sviluppo scheda interfaccia:**
  Durante questa fase si sviluppa una scheda per l'interazione uomo-macchina del singolo elettrodomestico.

- **WP6 - Progettazione e sviluppo scheda gestione:**
  L'obiettivo di questo modulo è la produzione di una scheda per la gestione dell'intero sistema, basata su microcontrollore ad alte prestazioni, che offre un'interfaccia utente centralizzata.

- **WP7 - Progettazione e sviluppo del software ad alto livello:**
  Questa fase prevede la realizzazione di un'infrastruttura software per l'implementazione di algoritmi che, partendo da una determinata base di conoscenza formata da informazioni eterogenee, permettano di supportare ed assistere chi si trova ad operare in cucina, disporre di un archivio remoto per la simulazione del potenziale supermercato e testare le capacità del sistema.

- **WP8 - Progettazione e sviluppo di un dimostratore:**
  In quest'ultima fase si effettua l'integrazione di tutti i prodotti progettati e realizzati durante il percorso progettuale, si verificano le loro potenzialità come sistema integrato, la corrispondenza tra le specifiche stilate durante il WP1 e le funzionalità espresse dal sistema e, infine, si individuano i

limiti dei singoli moduli in relazione al sistema complessivo, evidenziando i potenziali miglioramenti e gli sviluppi futuri.

3. Il modulo sviluppato: la control board

3.1 Introduzione: l'ARM7

Nella fase di sviluppo del firmware si è lavorato su due microcontrollori della Philips dotati di microprocessore ARM7TDMI-S, che fanno parte della famiglia di microprocessori general-purpose ARM, i quali offrono un ottimo rapporto fra prestazioni e consumo di potenza.

I microprocessori ARM sono basati su un'architettura di tipo RISC (*Reduced Instruction Set Computer*), la cui semplicità consente un elevato throughput di istruzione e un'eccellente risposta in tempo reale alle interruzioni.

Gli ARM7TDMI-S utilizzano la tecnologia pipelined a tre stadi: *Fetch*, *Decode* e *Execute*, per consentire al sistema di memorizzazione e di processing di lavorare ininterrottamente. Per quanto riguarda la struttura della memoria, essi sono basati su un'architettura di tipo Von Newmann con un unico bus dati a 32 bit per dati e istruzioni.

Una caratteristica peculiare di questa tipologia di microprocessori è la possibilità di gestire due set di istruzioni differenti: uno a 32 bit, detto *ARM Instruction Set*, e uno a 16 bit, detto *THUMB Instruction Set*, e permettono di impiegare entrambi i set nello stesso flusso di esecuzione. Il vantaggio nell'utilizzo di questa caratteristica sta nel fatto che la strategia Thumb implementa un set di istruzioni a 16 bit su un'architettura a 32 bit, offrendo così prestazioni maggiori rispetto alle architetture a 16 bit, con una densità di codice più elevata rispetto alle architetture a 32 bit.

In Figura 4 si riporta lo schema funzionale di un generico processore della famiglia ARM7:

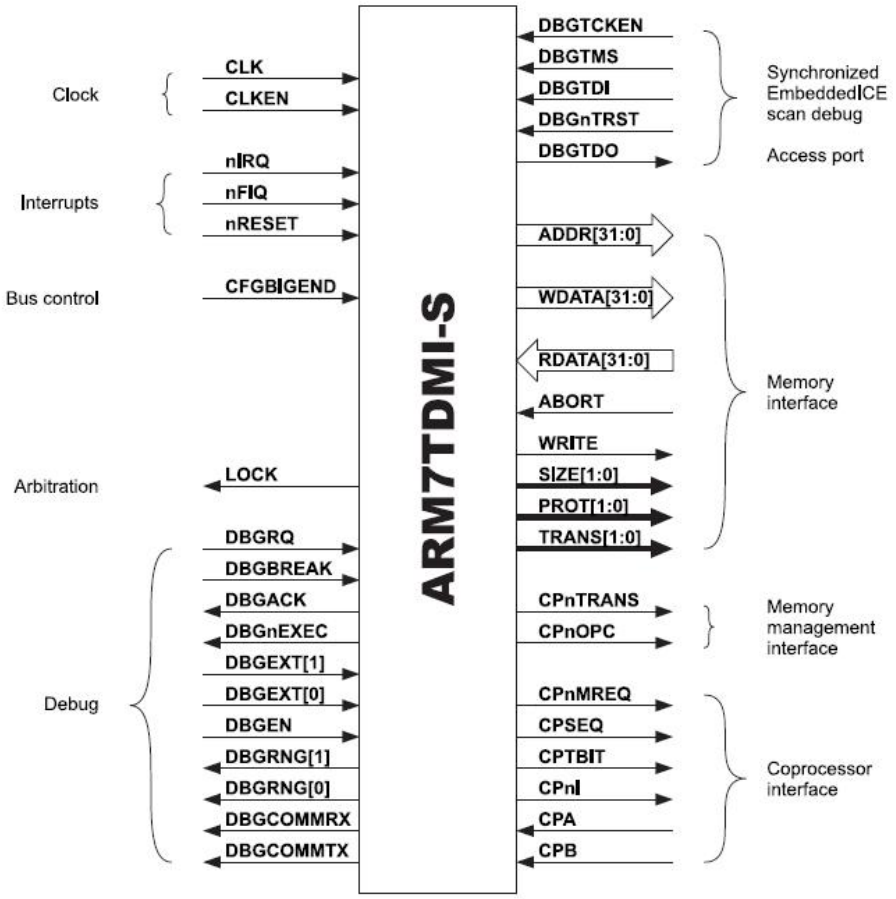

Figura 4: Schema funzionale di un generico processore della famiglia ARM7

3.2 Le piattaforme Embedded

Per lo sviluppo del protocollo Modbus è stata utilizzata la board della Olimex fornita con il KickStart Kit della IAR, sulla quale è montato il microcontrollore LPC2148 della Philips. Questa scelta è stata fatta poiché la scheda era già stata utilizzata dall'azienda per altri progetti: questo ha permesso di sfruttare il know-how acquisito e di ridurre il time-to-market.

Per la progettazione di AmicLan over TCP/IP e AmicLan over USB è stata invece utilizzata l'Evaluation Board MCB2300 della Keil, sulla quale è montato il microcontrollore LPC2378 della Philips.

Nel paragrafi seguenti si descriveranno le caratteristiche generali delle board utilizzate e si discuteranno le motivazioni che hanno portato alla scelta del secondo microcontrollore come ideale per lo sviluppo della scheda di controllo.

3.2.1 Philips LPC2148

Il LPC2148 è un microcontrollore basato sulla famiglia ARM7 pensato per applicazioni il cui requisito chiave è la miniaturizzazione, date le dimensioni e i consumi di potenza ridotti.

Le sue caratteristiche sono:

- Microprocessore a 16/32 bit della famiglia ARM7TDMI-S;
- 32 kB di RAM statica on-chip + 8kB condivisa con l'USB in DMA;
- 512 kB di memoria flash di programma on-chip;
- Programmazione In-System/In-Application (ISP/IAP) via bootloader on-chip;
- Interfacce EmbeddedICE RT e Embedded Trace per il debugging in tempo reale;
- Due convertitori A/D e un convertitore D/A a 10-bit;
- Due timer/counter a 32-bit, un'unità PWM e watchdog;
- RTC a basso consumo con alimentazione indipendente e ingresso di clock a 32 kHz dedicato;
- Controller di interrupt vectored con vettore degli indirizzi e priorità configurabile;
- Fino a 45 piedini Fast GPIO;
- Fino a 9 piedini di interrupt esterni sensibili al fronte e al livello;
- Interfacce seriali:
    - Due interfacce UART;
    - Due fast I2C-bus, SPI e SSP;

    - Controller USB 2.0 compliant con 2 kB (+8 kB condivisi) di RAM per gli endpoint;
- PLL on-chip che permette di ottenere frequenze di clock per la CPU fino a 60 MHz;
- PLL on-chip che permette di ottenere la frequenza di 48 MHz per l'USB;
- Oscillatore integrato on-chip operante con un quarzo esterno fra 1 MHz e 30 MHz o con un oscillatore esterno fino a 50 MHz;
- Due modalità di risparmio energetico: Idle e Power Down;
- Wake-up del processore dalla modalità Power-down via external interrupt, USB, Brown-Out Detect (BOD) o Real-Time Clock (RTC).

L'architettura del LPC2148 è costituita da una CPU della famiglia ARM7TDMI-S, da un bus locale per l'interfaccia con i controllori della memoria on-chip, da un bus AMBA di tipo AHB (Advanced High-performance Bus) per l'interfaccia con il controller di interrupt e da un bus di periferica VLSI (VPB, un superinsieme dell'AMBA Advanced Peripheral Bus per ARM) per la connessione alle funzioni di periferica on-chip (Figura 5). Il LPC2148 configura il processore ARM7TDMI-S per lavorare in modalità little-endian.

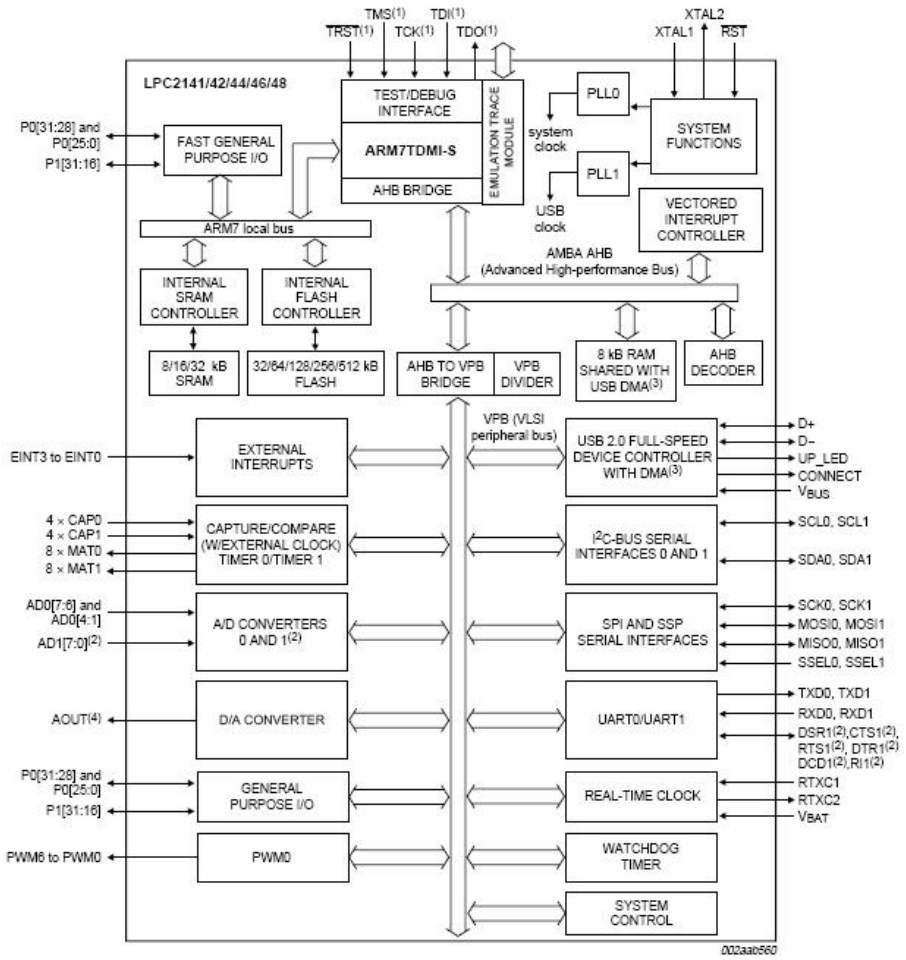

Figura 5: Schema a blocchi del microcontrollore LPC2148

3.2.2 Philips LPC2378

Il LPC2378 è un microcontrollore basato sulla famiglia ARM7 pensato per applicazioni che richiedono comunicazioni seriali per vari scopi, poiché incorpora controller per la gestione di diversi tipi di interfacce di comunicazione.

Le sue caratteristiche sono:

- Microprocessore a 16/32 bit della famiglia ARM7TDMI-S, che può lavorare fino ad una frequenza di 72 MHz;
- 32 kB di memoria RAM statica nel bus locale dell'ARM per accessi ad elevate prestazioni;

- 16 kB di memoria RAM statica per l'interfaccia Ethernet, utilizzabile anche come SRAM general purpose;
- 8 kB di memoria RAM statica per l'interfaccia USB, utilizzabile anche come SRAM general purpose;
- 512 kB di memoria flash di programma on-chip nel bus locale dell'ARM per accessi ad elevate prestazioni;
- Programmazione In-System/In-Application (ISP/IAP) via bootloader on-chip;
- Interfacce EmbeddedICE RT e Embedded Trace per il debugging in tempo reale;
- Sistema Dual AHB che permette quattro accessi simultanei DMA Ethernet, DMA USB e esecuzione del programma dalla memoria Flash on-chip senza competizione fra questi. Un bridge consente al DMA Ethernet di accedere agli altri sottosistemi AHB;
- Controller di memoria esterna che supporta dispositivi statici come Flash e SRAM. E' disponibile anche un bus parallelo di 8 bit per i dati e 16 bit per gli indirizzi;
- Controller di interrupt vectored avanzato che supporta fino a 32 interrupt vettorizzati;
- Controller AHB DMA general purpose (GPDMA) che può essere utilizzato con interfacce seriali SSP, con la porta I2S, con la porta per schede SD/MMC, nonché per trasferimenti da memoria a memoria;
- Interfacce seriali:
    - MAC Ethernet e controller DMA associato, che risiedono in un bus AHB indipendente;
    - Periferica USB 2.0 con layer fisico on-chip e controller DMA associato;

  - Quattro interfacce UART con generatore di baud rate frazionale, di cui una con supporto IrDA, che risiedono nel bus APB;
  - Due canali CAN, controller SPI, due controller SSP, tre interfacce I2C e una I2S su bus APB;
- Altre interfacce su APB:
  - Interfaccia memory card MMC e SD;
  - 104 piedini GPIO;
  - Convertitore A/D e D/A a 10 bit;
- Quattro Timer general-purpose;
- Un blocco PWM/Timer con supporto per il controllo di motori trifase;
- RTC con piedino di alimentazione separato. La sorgente di clock può essere l'oscillatore RTC o il clock dell'APB;
- 2 kB di SRAM alimentata dal piedino di alimentazione dell'RTC, che consente la memorizzazione dei dati quando il resto del chip non è alimentato;
- Watchdog Timer clockabile dall'oscillatore RC interno, dall'oscillatore RTC o dal clock dell'APB;
- Quattro modalità di risparmio energetico: Idle, Sleep, Power Down e Deep Power down;
- Quattro ingressi per gli interrupt esterni;
- Wake-up del processore dalla modalità Power-down via qualsiasi interrupt capace di operare in questa modalità;
- Due domini di potenza indipendenti, che consentono la regolazione del consumo di potenza in base alle caratteristiche necessarie;
- Oscillatore al quarzo on-chip con un range operativo da 1 MHz a 24 MHz;
- Oscillatore RC utilizzabile come clock di sistema;

- PLL on-chip che permette la generazione di due frequenze di clock distinte (CPU e USB).

L'architettura del LPC2378 consiste in una CPU della famiglia ARM7TDMI-S, in un bus locale per l'accesso ad alta velocità alla maggior parte della memoria on-chip, in un bus AMBA AHB (Advanced High-performance Bus), che consente l'interfacciamento con le periferiche ad alta velocità e con la memoria esterna, e infine in un bus AMBA APB (Advanced Peripheral Bus) per la connessione alle funzioni di periferica on-chip (Figura 6). Il microcontrollore configura il processore ARM7TDMI-S per lavorare in modalità little-endian.

Il LPC2378 implementa due bus AHB per consentire al blocco Ethernet di operare senza le interferenze causate dalle altre attività di sistema. L'AHB primario (AHB1) comprende il Vectored Interrupt Controller, il General Purpose DMA Controller, l'External Memory Controller, l'interfaccia USB e la SRAM da 8 kB, dedicata principalmente all'utilizzo con l'USB. L'AHB secondario (AHB2) comprende solo il blocco Ethernet e la SRAM associata. Inoltre, un bridge permette all'AHB2 di essere master nell'AHB1, consentendo così l'espansione del buffer dell'Ethernet nella memoria off-chip o nello spazio della memoria dell'AHB1 inutilizzato.

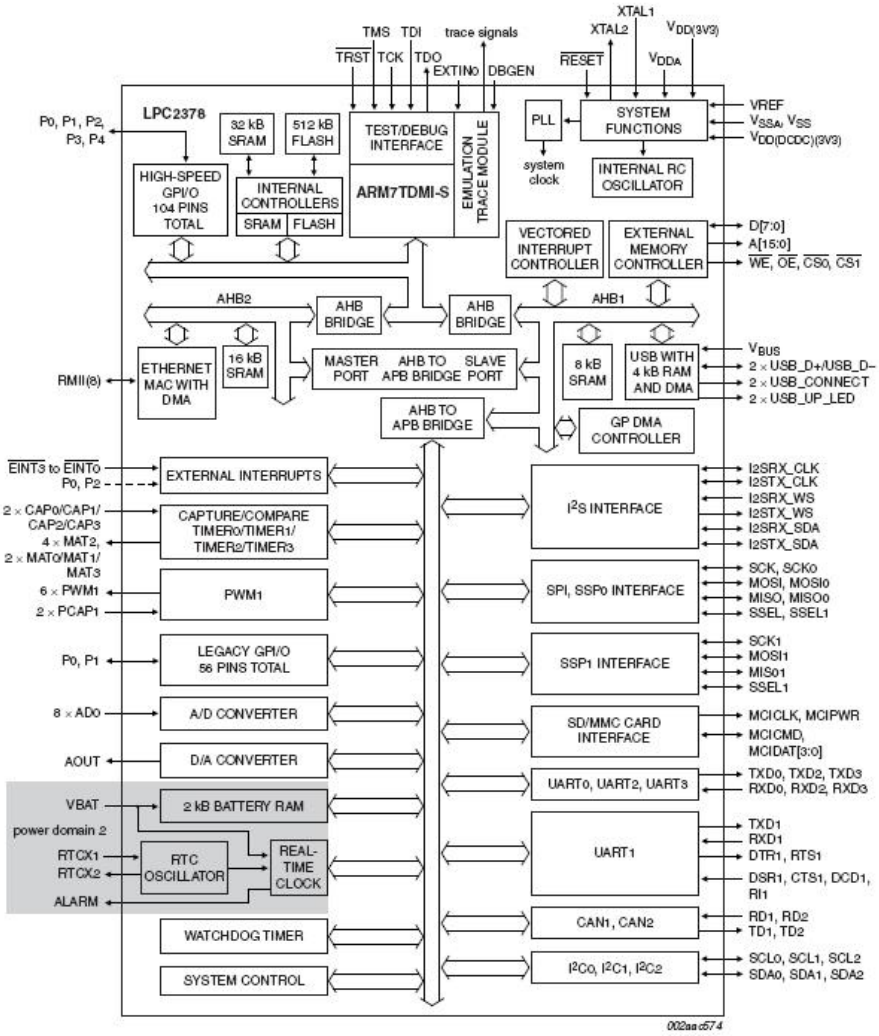

Figura 6: Schema a blocchi del microcontrollore LPC2378

3.2.3 Scelta del microcontrollore

In questo paragrafo si analizzeranno le differenze fra i due microcontrollori utilizzati, per comprendere il motivo dell'adozione del LPC2378 nello sviluppo della control board.

Poiché nelle specifiche della control board è prevista la possibilità di comunicare con il protocollo TCP/IP, si è avvertita subito l'esigenza di utilizzare un microcontrollore la cui architettura fosse dotata di MAC Ethernet: non essendo il LPC2148 provvisto di quest'interfaccia, esso non è adatto allo scopo.

Per quanto riguarda la capacità di memorizzazione, entrambi i microcontrollori sono dotati di memoria sufficiente a contenere sia il codice su Flash di programma, sia i buffer di supporto all'esecuzione; non va comunque dimenticato che, da specifiche, il progetto ELIK è pensato anche per offrire un substrato hardware e firmware per applicazioni future. La possibilità del LPC2378 di accedere ad alta velocità a un'eventuale memoria esterna, quindi, è sicuramente un fattore da non trascurare.

Inoltre, per lo sviluppo della scheda di controllo, è necessario un microcontrollore che permetta la gestione ottimale delle temporizzazioni, dato che la gestione dei vincoli di tempo è critica in quasi tutti i protocolli che si andranno a sviluppare. Il LPC2378 è indubbiamente preferibile, essendo dotato di quattro blocchi Timer/Counter contro i due del LPC2148.

La scheda di controllo deve poi essere in grado di gestire più interfacce contemporaneamente, nel rispetto delle temporizzazioni previste dai protocolli. Quindi è auspicabile che il microcontrollore gestisca almeno le interfacce ad alta velocità nel modo più performante possibile: il LPC2378 accede al controller USB via bus AHB, a differenza del precedente che vi accede via bus APB, quindi con prestazioni sicuramente peggiori rispetto al primo.

Infine, un fattore non marginale nello sviluppo di un sistema embedded è il *power control*: il LPC2378 offre maggiori funzionalità per la gestione del risparmio energetico rispetto al LPC2148, fra cui quattro modalità di funzionamento, contro le due del microcontrollore precedente, e l'utilizzo di due *power domain* differenti.

Per concludere, l'unico punto di debolezza del LPC2378 è che sono presenti pochi esempi e molti bachi ancora non documentati, motivi che hanno reso la sua programmazione più difficoltosa, dilatando così il time-to-market.

Tutte queste considerazioni hanno portato alla conclusione che i vantaggi in termini di prestazioni del LPC2378 rispetto al LPC2148 e simili sono sufficienti a considerare trascurabile il rallentamento nello sviluppo.

3.3 Il progetto hardware

3.3.1 Protel 99 SE

Per lo sviluppo del PCB si è utilizzato il sistema di progettazione elettronica **Protel 99 SE** di Altium. Esso, sebbene ormai soppiantato da altri applicativi più potenti, permette ancora di raggiungere efficacemente gli obiettivi dell'azienda e quindi, visto che ormai rientra nel bagaglio di conoscenza della stessa e che ancora si producono librerie aggiornate per questo software, non si ritiene conveniente il passaggio a sistemi più evoluti.

Questo applicativo offre una suite di strumenti integrati completa, che segue lo sviluppo dal progetto ad alto livello fino al layout della scheda. La caratteristica peculiare di questo sistema è che tutti gli strumenti sono accessibili da un unico ambiente di sviluppo, il *Design Explorer*, e questo permette di avere sempre sotto controllo l'intero progetto.

Il tool permette di compilare un circuito per l'implementazione in un PLD (programmable logic device) e di impostare simulazioni miste analogico/digitale direttamente dallo schematico (quest'ultima possibilità grazie all'utilizzo di librerie di componenti di simulazione). Inoltre, Protel 99 SE fornisce una libreria di schematici e footprint PCB molto estesa e aggiornabile accedendo al portale web.

L'applicativo include anche fogli di calcolo, editor di testo e tool di creazione di macro integrati, consentendo così di gestire tutti gli aspetti di progetto senza abbandonare il *Design Explorer*.

Quest'ultimo è strutturato secondo un'architettura client/server, cosa che consente di estendere le funzionalità del tool con Add-On reperibili dal sito dell'Altium o da terzi.

3.3.2 Schema a blocchi

In Figura 7 è mostrato lo schema a blocchi dei componenti che caratterizzano l'architettura hardware della scheda di controllo.

Il blocco fondamentale, sul quale verteranno buona parte delle scelte architetturali relative al PCB, è quello relativo al microcontrollore ARM7, denominato *Micro_ARM7*. In Figura 7, per il blocco in esame, sono messi in evidenza i piedini di I/O utilizzati per le interconnessioni con le altre periferiche, nonché per il collegamento con componenti utili al funzionamento del microcontrollore.

Il blocco *Serial Ports* rappresenta i pin di interfaccia per la gestione delle periferiche RS232 e RS485: si è dotato il sistema di una porta RS232 e due porte multiplate RS232/RS485.

Il blocco *Ethernet* mostra l'interfaccia del modulo PHY che implementa lo strato fisico del protocollo Ethernet, che opererà quindi come gateway fra microcontrollore e porta RJ-45.

Il blocco *Power_supply* rappresenta l'insieme dei dispositivi che hanno il compito di fornire alla scheda i requisiti di potenza richiesti; vedremo che questo blocco sarà composto da un alimentatore switching e da un alimentatore lineare.

Il blocco *Triacs_latch* rappresenta una serie di latch che permettono di pilotare dei triacs, utili per il controllo dei comandi di tipo ON/OFF dell'elettrodomestico che sarà pilotato dal PCB.

Nello schema a blocchi non è riportata l'interfaccia USB perché la si è considerata come parte del blocco *Micro_ARM7*.

Per alcuni blocchi non è riportato lo schema elettrico per ragioni di riservatezza imposte dall'azienda. Il blocco *Isolated_DI* rappresenta

una serie di ingressi digitali isolati galvanicamente, *Connettore* è il blocco di interfaccia, i blocchi *Isolated_probes_n* sono delle sonde di misurazione del livello di liquidi in serbatoi e i blocchi *Thermon* sono dei sensori di temperatura. Inoltre, in figura, sono riportati anche i triac nei blocchi *TriacXX_n*.

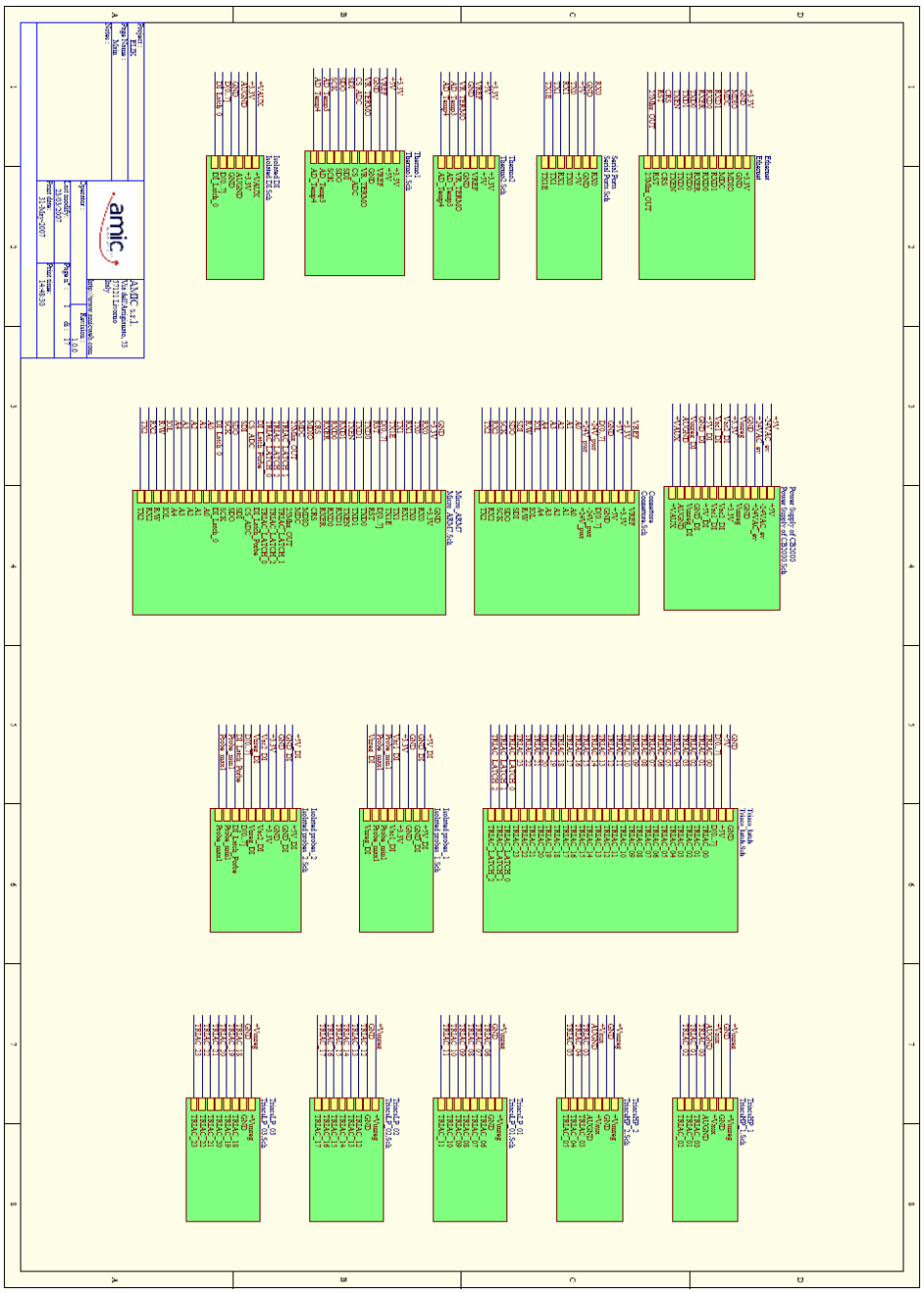

Figura 7: Schema a blocchi dell'architettura hardware della scheda di controllo

3.3.3 Schema elettrico

In questo paragrafo verranno descritte in dettaglio le scelte circuitali fatte per i singoli blocchi visti nel paragrafo precedente. Per una maggiore comprensione degli schemi elettrici mostrati di seguito, si ritiene utile sottolineare che i pin di collegamento fra blocchi distinti vengono mostrati in figura collegati a box gialli, etichettati con un nome comune ai blocchi interagenti.

3.3.3.1 Blocco Micro_ARM7

In Figura 8 è mostrato lo schema elettrico relativo ai moduli circuitali utili per il corretto funzionamento del microcontrollore LPC2378, nonché lo schema elettrico relativo alla periferica USB, all'interfaccia JTAG e ad un modulo per il collegamento di una memoria Flash esterna.

Nella parte destra della figura è riportato il package del LPC2378. Ai quattro lati del package sono rappresentati dei filtri passa-basso, costituiti da quattro capacità in parallelo, che permettono lo smorzamento di eventuali rumori dovuti ad interferenze elettromagnetiche sulla linea di alimentazione. Al pin **RESET** è collegato un jumper, che permette di avviare la fase di reset del microcontrollore. Quest'ultima riporta le porte di I/O e le periferiche nello stato di default e memorizza, nel registro Program Counter, l'indirizzo 0x0. Ai pin **XTAL1** e **XTAL2** è collegato il circuito oscillatore relativo al *main oscillator*, caratterizzato da un quarzo alla frequenza di 12 MHz.

Ai pin **RTCX1** e **RTCX2** è collegato, invece, il circuito oscillatore relativo al *Real Time Clock*, caratterizzato da un quarzo alla frequenza di 32.768 KHz.

Al pin **VBAT** è collegato il circuito riportato sopra il package, che permette di alimentare il PCB non solo tramite collegamento alla linea elettrica, ma anche per mezzo di batterie.

Il package **M25P10** è una memoria Flash seriale a 1 Mbit (128K x 8) collegata ai pin relativi all'interfaccia SSP (Synchronous Serial Port) del microcontrollore.

Il componente **2X10P** è un header femmina con 10 pin per riga, che costituisce la TAP (test access port) per la connessione con dispositivi di comunicazione JTAG standard. A questo sono collegati i pin relativi al controllo dell'interfaccia JTAG del microcontrollore.

L'ultimo frammento di circuito da analizzare è quello che permette la connessione della porta USB ai relativi pin del microcontrollore. Poiché il protocollo fisico è caratterizzato dal pilotaggio di due sole linee, lo schema elettrico è molto semplice e non necessita di una descrizione approfondita.

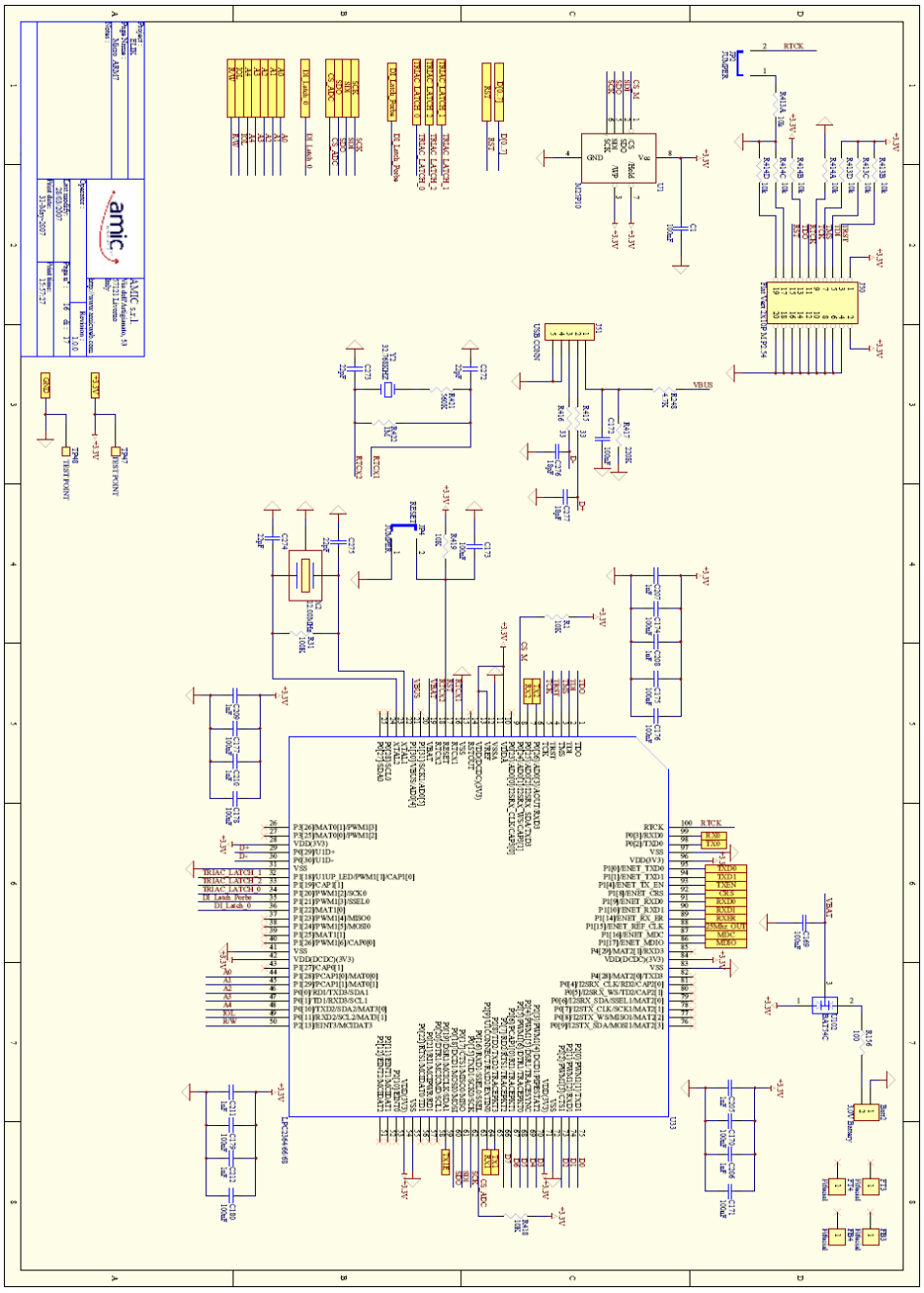

Figura 8: Schema elettrico del modulo Micro_ARM7

3.3.3.2 Il blocco Triacs_latch

Questo blocco è costituito da 3 componenti **74HC573**: si tratta di D-latch trasparenti, che offrono otto ingressi di tipo D per ogni latch e altrettante uscite 3-state. Questi componenti vengono utilizzati per pilotare 24 thyristor di tipo triac. I triac sono dispositivi a tre terminali, progettati per controllare carichi in corrente alternata. Due dei terminali sono detti *anodi* e costituiscono la via di passaggio per la corrente controllata, mentre il terzo, definito *gate*, è l'ingresso di controllo.

Nel progetto in questione i triac vengono utilizzati come relè a semiconduttore: in questo caso è sufficiente mantenere sul gate una corrente di attivazione per tutto il periodo che si desidera tenere attivo il carico. Questo spiega la necessità di includere dei latch fra i pin del microcontrollore, che serviranno per selezionare lo stato di funzionamento e i thyristor.

Nel package 74HC573 i pin hanno le seguenti funzionalità:

- D0 – D7 : ingressi dati;
- LE : ingressi di abilitazione dei latch (attivo alto);
- OE : ingressi di abilitazione delle uscite 3 – state (attivo basso);
- GND : ground;
- Q0 – Q7 : uscite 3 – state dei latch;
- VCC : tensione di alimentazione.

In Figura 9 è riportato lo schema elettrico relativo al blocco in questione.

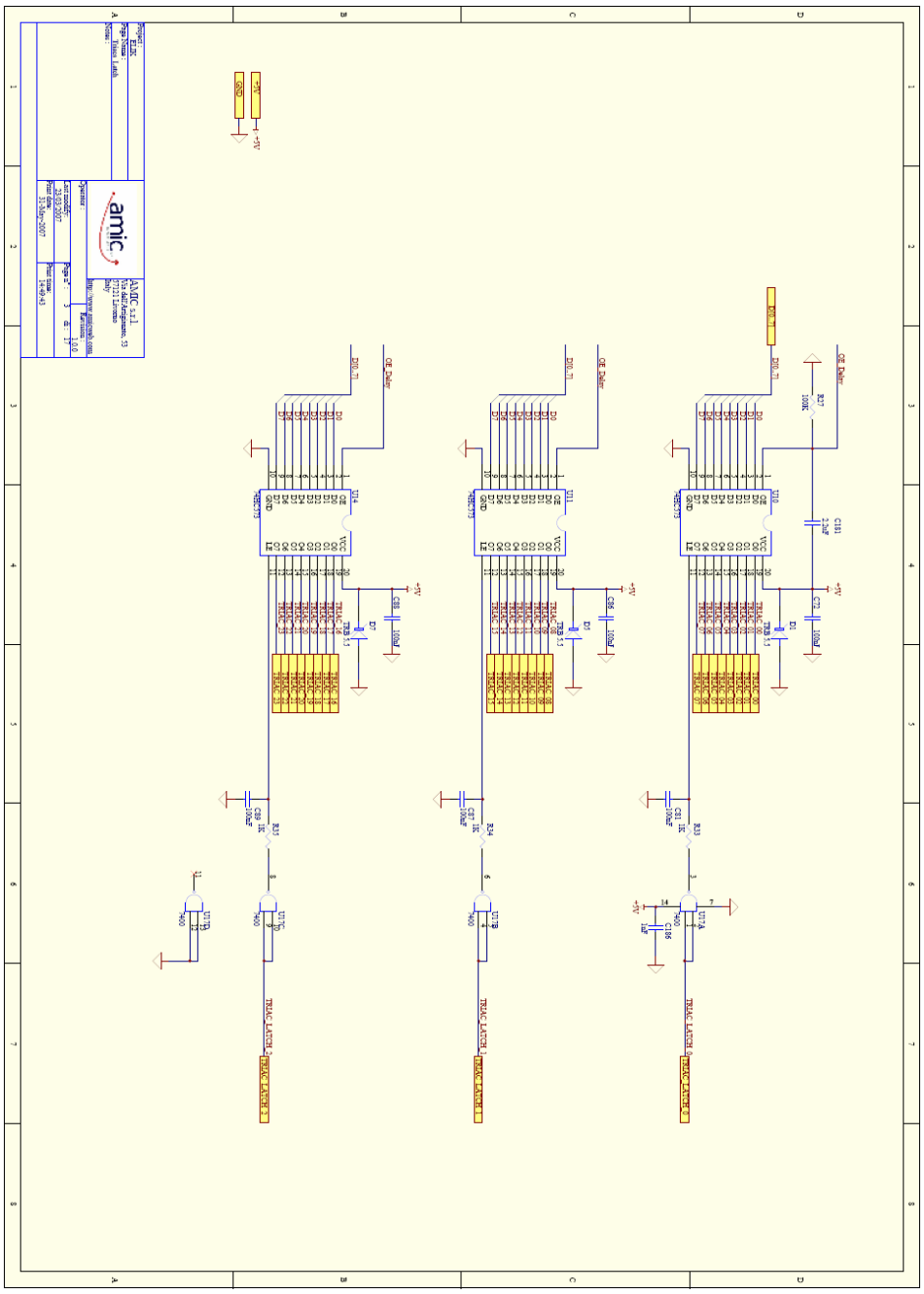

Figura 9: Schema elettrico del modulo Triacs_latch

3.3.3.3 Il blocco Power_Supply

Il blocco in oggetto è costituito da due sottounità circuitali che implementano, rispettivamente, un alimentatore switching (in alto in Figura 10) e un'alimentatore lineare (in basso nella stessa figura).

L'alimentatore lineare è composto da un trasformatore, un ponte di Graetz per l'ottenimento di una tensione positiva, dei condensatori di filtraggio e l'elemento di regolazione in tensione **uA78M05**. Questo circuito integrato mantiene costante la tensione in uscita e può generare una corrente in uscita fino a 500 mA. Dallo stesso trasformatore si genera anche una tensione non regolata +VAUX.

L'alimentatore switching è composto da un trasformatore, un ponte di Graetz, dei condensatori di filtraggio e il regolatore **LM2596**. Questo regolatore monolitico integrato offre tutte le funzioni attive per un regolatore switching step-down, capace di pilotare un carico di 3A e fornendo in uscita una tensione costante di 3.3V. Lavorando a 150 kHz esso consente l'utilizzo di componenti di filtraggio di dimensioni ridotte. Al regolatore switching segue la squadra LC, necessaria per il filtraggio da ondulazioni e disturbi della sua uscita.

Dallo stesso alimentatore switching viene inoltre generata, a partire dalla tensione Vunreg, una tensione di 5V, per mezzo di un altro regolatore di tensione **uA78M05** (a destra in Figura 10).

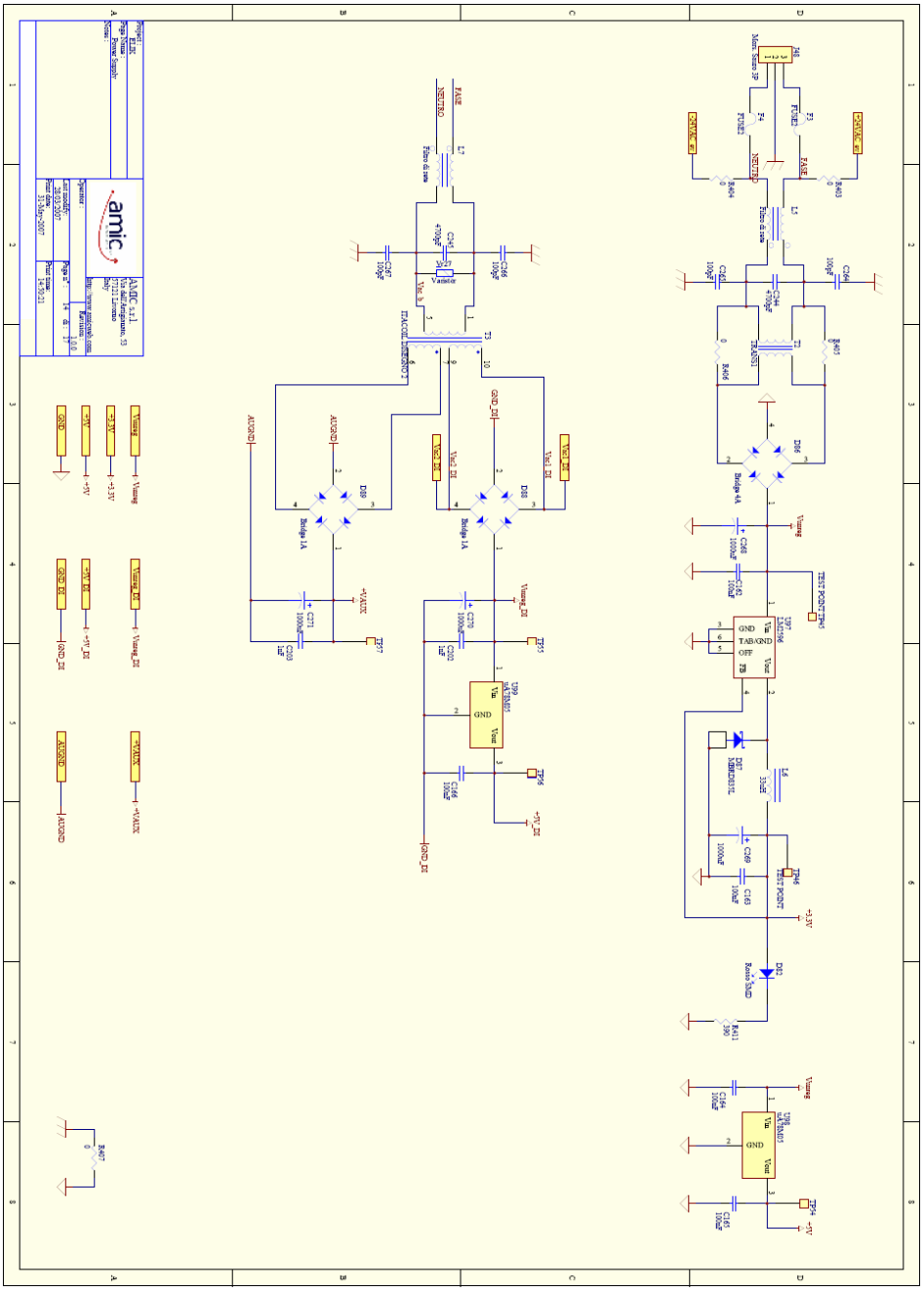

Figura 10: Schema elettrico del modulo Power Supply

3.3.3.4 Il blocco Serial_Ports

Il blocco relativo alle porte seriali è costituito da due sottoinsiemi circuitali, ciascuno relativo ai due tipi di interfaccia RS232 e RS485. Il blocco permetterà di ottenere il corretto pilotaggio di due porte seriali: una di tipo RS232 e una multiplexata RS232/RS485.

Il circuito nella parte superiore della Figura 11 è quello che consente il trasferimento seriale sull'interfaccia RS485. Il componente principale è il transceiver **SN75HVD**, che combina un driver di linea differenziale 3-state con un ricevitore di linea di ingresso differenziale. Questo circuito integrato è progettato per comunicazioni bidirezionali su linee di trasmissione multipunto bilanciate che seguono lo standard TIA/EIA-485-A.

Nel package SN75HVD i pin hanno le seguenti funzionalità:

- /RE : ingresso di abilitazione della ricezione (attivo basso);
- DE : ingresso di abilitazione dell'ingresso (attivo alto);
- RO : uscita del ricevitore;
- DI : ingresso del driver;
- VCC, GND : tensione di alimentazione, ground;
- A, B: ingressi del ricevitore.

Il circuito a valle del transceiver permette di regolare i valori di tensione e corrente in uscita dalla porta, in funzione delle specifiche del transceiver stesso.

Il circuito nella parte inferiore è, invece, quello che consente il trasferimento seriale sull'interfaccia RS232. Il componente principale è il transceiver **MAX202**, che rende possibile la conversione delle tensioni a 5V degli ingressi nelle tensioni a ±10V richieste per generare i livelli delle uscite previsti dallo standard EIA/TIA-232E.

Nel package MAX202 i pin hanno le seguenti funzionalità:

- C1, C2, V: questi sei pin realizzano il convertitore *charge-pump voltage*, che permette la conversione da 5V a

±10V. Il primo usa il condensatore C57 per raddoppiare la tensione, memorizzando i 10V nel condensatore C49 (V+); il secondo inverte la tensione e la memorizza nel capacitore C50 (V-);

- R1i, R2i, R1o, R2o: ingressi (Rni) e uscite (Rno) del transceiver relativi alla ricezione;
- T1i, T2i, T1o, T2o: ingressi (Tni) e uscite (Tno) del transceiver relativi alla trasmissione;
- VCC, GND : tensione di alimentazione, ground;

Per ottenere il multiplexing delle interfacce vengono utilizzati due diodi shottky e la linea RX1 viene mantenuta alta: si è scelta questa configurazione dato che i ricevitori di entrambe le interfacce hanno lo stato di riposo a 5V.

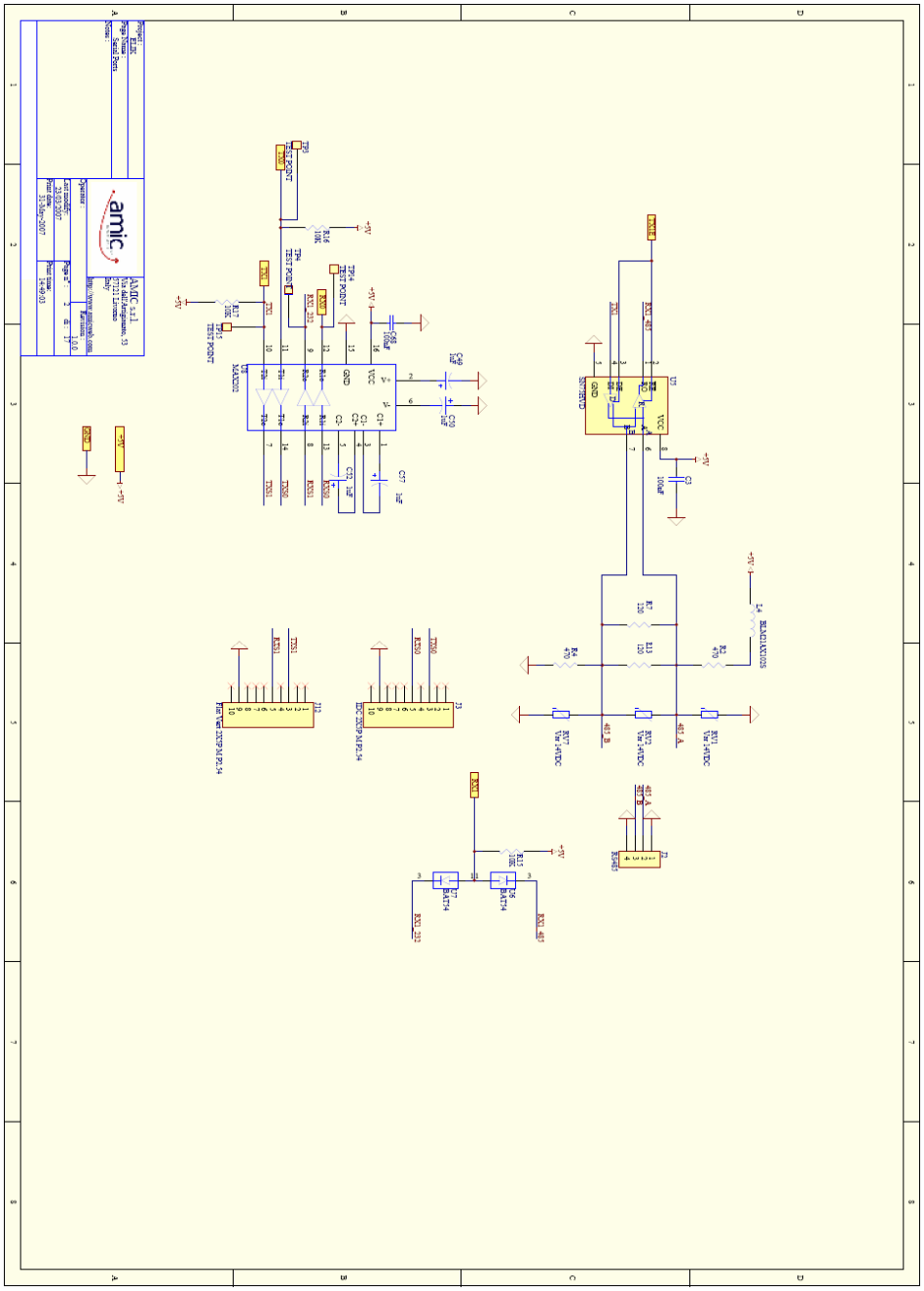

Figura 11: Schema elettrico del modulo Serial Ports

3.3.3.5 Il blocco Ethernet

Questo blocco è costituito da una porta RJ-45 e dal package **DP83848C**, che realizza il livello fisico (PHY) del protocollo Ethernet. Lo schema elettrico in questione è riportato in Figura 12.

Il modulo DP83848C incorpora le interfacce MMI (Media Indipendent Interface), RMII (Reduced Media Indipendent Interface) e SNI (Serial Network Interface). Per una descrizione dettagliata delle caratteristiche delle diverse interfacce si rimanda alle specifiche relative.

Per questo progetto si è scelto di sfruttare l'interfaccia RMII e, per fare ciò, è necessario impostare i pin MII_MODE e SNI_MODE rispettivamente al valore logico 1 e 0 in fase di strap. Quindi al pin MII_MODE si è collegata una resistenza di pull-up e il pin SNI_MODE è stato posto in alta impedenza, essendo esso già collegato ad una resistenza di pull-down interna. Per questa modalità di funzionamento è previsto l'utilizzo di un oscillatore a 50 MHz collegato ai pin X1 e X2.

Per filtrare i rumori causati da interferenze elettromagnetiche sull'alimentazione sono stati utilizzati tre condensatori da 100 nF ciascuno.

Al pin Power Feedback Output (PFBOUT) sono collegati, come consigliato dal datasheet, un condensatore a 100nF e un condensatore al tantalio a 10 uF in parallelo.

Ai pin Power Feedback Input (PFBIN) è connesso, molto vicino al pin, un condensatore da 100 nF, oltre al circuito caratteristico del PFBOUT.

Ai pin LED_LNK, LED_SPEED e LED_RX sono collegati tre diodi SMD rossi che indicano, rispettivamente, lo stato del link, la velocità (se ON la velocità è 100Mbps, se OFF è 10Mbps) e l'attività del PHY.

Il circuito a destra, in Figura 12, mostra i componenti utilizzati per il collegamento con la porta RJ-45 e le loro interconnessioni.

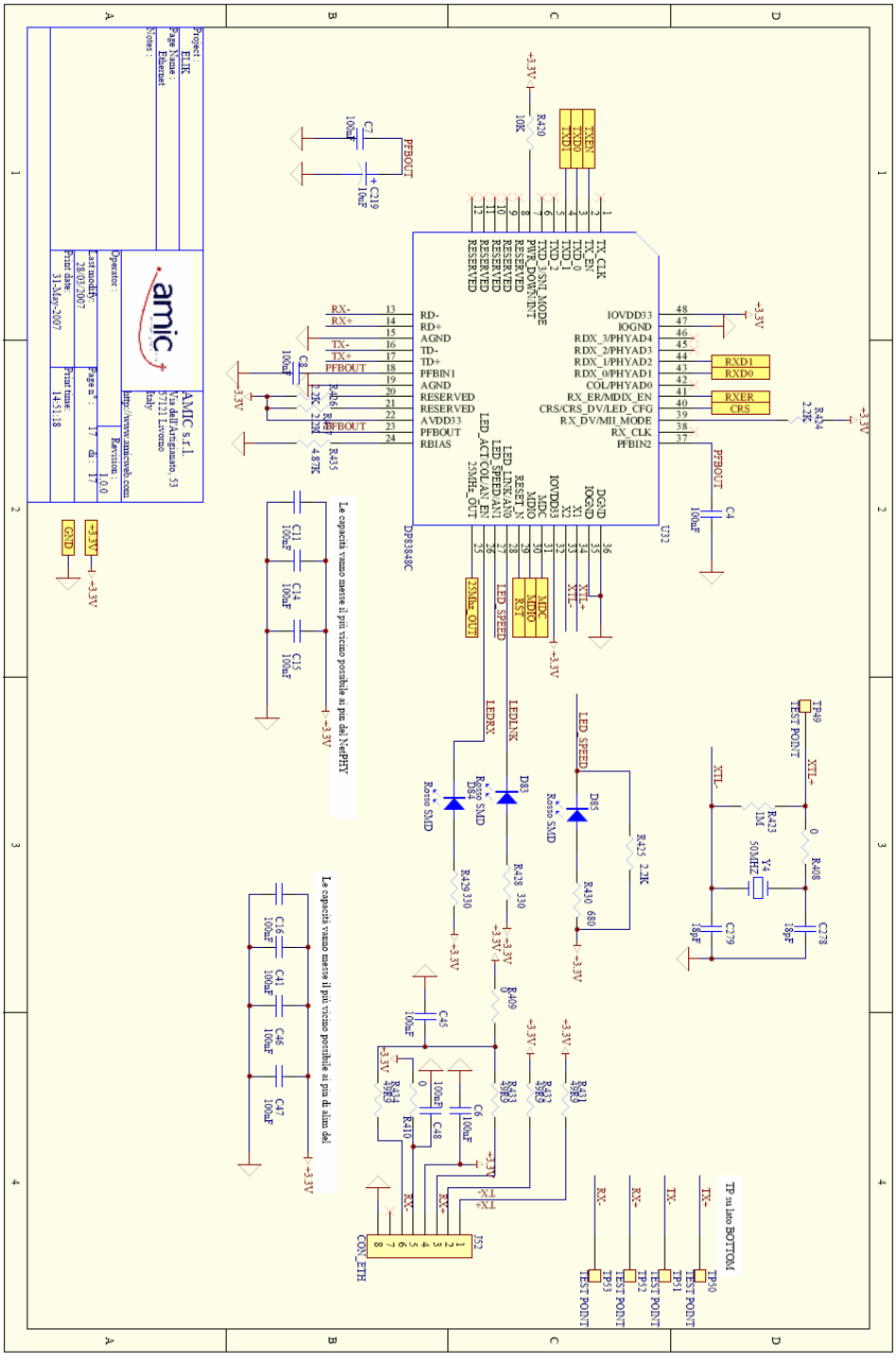

Figura 12: Schema elettrico del modulo Ethernet

3.4 Il firmware

3.4.1 I tool e gli ambienti di sviluppo

3.4.1.1 Introduzione

La scelta del tool di sviluppo è sicuramente una delle fasi più critiche nello sviluppo dell'infrastruttura firmware di un microcontrollore: da esso dipende infatti, oltre che la capacità di supporto nello sviluppo di codice corretto e funzionale, anche il suo livello di ottimizzazione, e quindi ha un peso notevole nelle prestazioni.

Un ambiente di sviluppo completo può essere suddiviso in tre strati, ognuno avente funzionalità diversa: uno strato di supporto hardware, un motore di creazione del codice macchina, detto toolchain, e una GUI; ad eccezione del layer intermedio, che comprende l'insieme dei componenti che permettono la compilazione, il linking e il building del file esadecimale che verrà caricato nella memoria del microcontrollore, gli altri due layer possono offrire funzionalità più o meno avanzate, o addirittura possono non essere presenti.

Lo strato di supporto hardware è quello che si occupa dell'interfacciamento della scheda con il calcolatore utilizzato per lo sviluppo. L'unico ruolo indispensabile di questo layer è quello di consentire il trasferimento dell'esadecimale nella memoria della scheda, ma può offrire anche supporto per il debugging e testing del firmware. Esistono fondamentalmente due metodi di accesso alla memoria: tramite porta seriale o utilizzando lo standard JTAG. Il primo metodo consente solamente la lettura e la scrittura della memoria, mentre il secondo permette l'accesso alla maggior parte delle risorse offerte dalla scheda anche durante l'esecuzione del codice, grazie alla definizione di una porta di accesso standard per il test (interfaccia JTAG) e di un'architettura *boundary-scan*, secondo lo standard **IEEE 1149.1**. La descrizione del protocollo e della

circuiteria di questo standard esula dagli scopi della trattazione: per una trattazione esaustiva si rimanda quindi alle specifiche IEEE.

La toolchain è costituita da una serie di componenti che permettono, mediante la loro cooperazione, di generare il file adatto al microprocessore in uso, a partire dal codice scritto nel linguaggio prescelto. I criteri di scelta di questo strato si baseranno quindi sul linguaggio di programmazione che si vuole utilizzare e sul tipo di macchina per la quale si vuole generare l'immagine della memoria; inoltre, un ulteriore fattore da non trascurare è la capacità del tool (in particolare del compilatore) di consentire l'ottenimento di codice che sfrutti le risorse disponibili nel modo più efficiente e ottimizzato possibile.

L'interfaccia grafica (GUI) è utile per ridurre il lavoro intellettuale del progettista e migliorarne l'efficienza, poiché consente una gestione complessiva del progetto più agevole, riducendo così i rischi di errore umano nello sviluppo.

La scelta del tool di sviluppo dovrà quindi tener conto di questi tre strati e delle caratteristiche offerte da ciascuno di essi, in modo da poter individuare una soluzione che raggiunga un ottimo compromesso fra costo del tool, efficienza ed efficacia delle sue componenti e facilità d'uso.

3.4.1.2 La scelta del tool

Durante lo sviluppo del progetto ELIK si è avuta la possibilità di confrontare diversi ambienti di sviluppo, alla ricerca di quello che più rispondesse alle esigenze dell'azienda e del progetto stesso in merito a costi e prestazioni. Per poter analizzare i singoli IDE a confronto è necessario inquadrare, preliminarmente, quali sono i requisiti richiesti all'ambiente di sviluppo.

Il primo requisito è che i costi siano i più limitati possibile, affinchè non vadano a gravare sui costi NRE per quanto riguarda il progetto ELIK e che, per progetti futuri commissionati da aziende esterne, non le costringano a spese eccessive che potrebbero dissuaderle dall'incaricare l'Amic S.r.l. dello sviluppo degli stessi.
Il secondo requisito è che il tool sia utilizzabile anche per lo sviluppo su microcontrollori diversi da quello scelto; questo sia per permettere di sfruttare il know-how acquisito anche per progetti futuri, sia per non dover acquistare ambienti di sviluppo diversi ad ogni progetto proposto.
Un ulteriore requisito è che l'ambiente di sviluppo sia dotato di interfaccia grafica, perché più persone potrebbero collaborare alla progettazione del firmware e quindi è fondamentale ridurre la complessità del progetto, offrendo agli sviluppatori un ambiente user-friendly.
Tenendo conto dei requisiti sopra citati, si analizzeranno ora gli ambienti di sviluppo provati, mettendo in evidenza i punti critici di ognuno di essi.
Il primo ambiente di sviluppo preso in esame è l'**IAR Embedded Workbench** della *IAR Systems*. Questo IDE offre un ambiente integrato che fornisce allo sviluppatore un'interfaccia grafica molto semplice e intuitiva; il suo costo si può collocare nella fascia medio/alta. Il framework è formato dai seguenti componenti:

- **ARM IAR C/C++ Compiler**;
- **ARM IAR Assembler**;
- **IAR XLINK Linker;**
- **IAR XAR Library Builder e IAR XLIB Librarian**;
- Un editor;
- Un project manager;
- Un'utilità di build da riga di comando;

- **IAR C-SPY® debugger**: un debugger con linguaggio di alto livello.

Su questo IDE è stata sviluppata la prima versione del protocollo ModBus per il microcontrollore LPC2148, quindi si sono potute valutare appieno le sue potenzialità. L'interfaccia grafica è molto intuitiva e non ha bisogno di uno studio approfondito sul suo funzionamento; inoltre, il sistema è dotato di un project manager che consente la gestione del progetto in maniera efficace. Il primo difetto che si è riscontrato è che il framework si basa su componenti proprietari e perciò, sebbene esso sia facile da usare, è necessario uno studio preliminare del compilatore per poter sfruttare appieno le sue potenzialità; inoltre, il know-how derivante è vincolato all'ambiente in questione. Un'ulteriore limitazione sta nel fatto che il compilatore permette di sviluppare firmware solo per microcontrollori basati su architettura ARM.
Dunque questo ambiente, nonostante soddisfi eccellentemente il terzo requisito, non si può ritenere accettabile per quanto riguarda costi e flessibilità.
Il secondo ambiente di sviluppo valutato è **uVision 3** della *Keil*. Questo è sicuramente uno fra gli ambienti di sviluppo per microcontrollori più diffusi e potenti presenti nel mercato, per via della sua flessibilità sia in termini di microcontrollore sia in termini di toolchain. Il framework è formato dai seguenti componenti di base:

- **uVision IDE;**
- **RealView C/C++ Compiler;**
- **RealView Macro Assembler;**
- **RealView Utilities;**
- **RL-ARM RealTime Library;**
- **uVision Debugger.**

Anche l'ambiente di sviluppo della Keil offre un compilatore proprietario, ma ha a suo favore due fattori importanti: il primo è che l'ARM Co. ha adottato il RealView come standard per la programmazione dei microcontrollori di questa famiglia; il secondo è che questo framework consente l'utilizzo di toolchain differenti, pur lavorando sullo stesso IDE. Altro punto a favore di questo ambiente è che consente la programmazione sulla maggior parte dei microcontrollori presenti nel mercato. Si può quindi affermare con certezza che esso soddisfa appieno il secondo requisito. Anche il terzo requisito è pienamente raggiunto, vista la semplicità d'uso della piattaforma di sviluppo uVision 3. L'unico fattore negativo di questo framework è che rientra nella fascia di costo alta, pur non acquistando il modulo RealView. Per questo motivo si è deciso di valutare delle alternative a prezzo più basso, non escludendo comunque la possibilità di un investimento futuro su questo prodotto.

L'ultimo ambiente di sviluppo preso in esame è **WinIDEA** della *iSystem*. Questo IDE offre un'interfaccia grafica non di qualità ottima come i precedenti ma notevolmente usabile, soprattutto se si tiene conto del suo basso costo. Anche questo, come i precedenti, offre assieme all'IDE un adattatore USB/JTAG che consente il debugging on-chip e la programmazione della memoria. Inoltre, così come il framework della Keil, anche quello della iSystem permette di scegliere una toolchain differente da quella proposta dalla casa. In virtù di tutte queste caratteristiche, e visto che rispetta tutti i requisiti richiesti, si è scelto di adottare quest'ultimo come ambiente di sviluppo per il progetto ELIK. Per soddisfare appieno il secondo requisito si è deciso comunque di non sfruttare la toolchain offerta dalla iSystem, ma di utilizzare la toolchain del framework opensource **WinARM**, le cui componenti sono:

- **Compilatore GNU-C/C++ versione 4.0.2**: cross compiler/linker/assembler con inclusa la *stdlib3*. La

configurazione supporta ARM-Mode, Thumb-Mode e miste ARM/Thumb – Mode, little/big-endian e emulazione floating point;

- **GNU-Binutils versione 2.16**: CVS snapshot 13.1.2006;
- **newlib versione 1.14.0**: build per syscall rientranti;
- **newlib-lpc Rel.5**: syscall rientranti per newlib e Philips LPC;
- **GNU-Utils**: utilità di supporto per il compilatore/linker (make, sh, etc. tutte Win32-native);
- **ARM header-files**: definizione dei registri da gnuarm.org;
- **Applicazioni di esempio** con codici sorgenti completi, makefile, script per il linker e codici di startup per i microcontrollori con core ARM7TDMI Philips LPC2000 e Atmel AT91SAM7;
- **Lpc2lisp versione 1.31**: in-system-programming software per le famiglie Philips LPC2xxx e Analog Devices ADUC 70xx scritto da Martin Maurer. Include anche la versione beta 1.33;
- **GDB Versione 6.3.50**: Debugger da riga di comando della Codesourcery.

Questa scelta permette di non vincolarsi eccessivamente all'IDE, che comunque ha un prezzo, per quanto basso. Si è pertanto liberi di utilizzare, per eventuali progetti futuri, il tool di sviluppo WinARM anche su IDE diversi (se l'ambiente lo permette) o di non utilizzarne affatto e sfruttare solo le componenti offerte dal framework WinARM.

3.4.2 Inizializzazione della scheda

La fase di startup e inizializzazione della scheda è sicuramente una delle fasi più critiche di tutto lo sviluppo del firmware per un microcontrollore: infatti, da questa dipende il corretto funzionamento di tutte le periferiche e del microcontrollore stesso, poiché è qui che

vengono impostate le frequenze di clock, abilitate e eventualmente configurate le periferiche, eccetera.

Il codice sorgente che implementa le procedure caratteristiche di questa fase è generalmente diviso in tre parti: un file script per il linker, un file di startup scritto in codice assembler e un file di inizializzazione scritto in linguaggio C.

Lo script per il linker contiene essenzialmente informazioni sulla distribuzione della memoria.

Nella prima parte vengono definiti gli spazi di memoria: ad esempio, viene definito l'intervallo di indirizzi destinato a contenere gli stack, lo spazio di memoria logica in cui è mappata la flash, quello in cui è mappata la SRAM, e così via. In Figura 13 è riportato il layout della memoria scelto per il progetto ELIK.

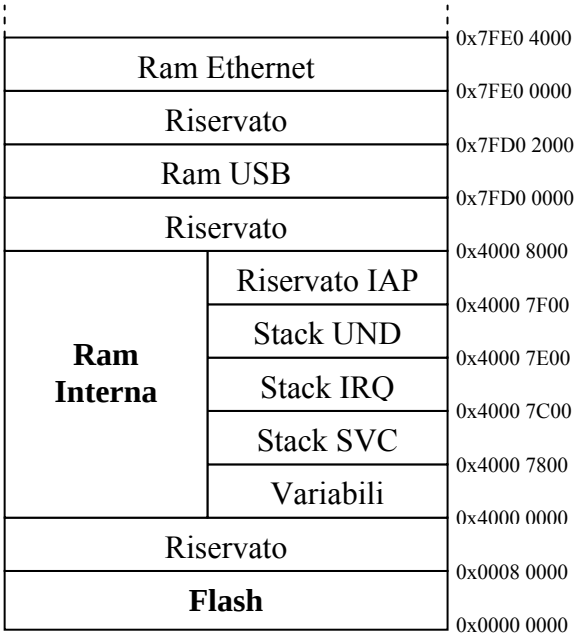

Figura 13: Layout della memoria scelto per il progetto ELIK

Le modalità operative del microprocessore ARM7 sono sette, quindi sarebbe stato necessario allocare lo spazio per tutte le modalità: dal momento che si è scelto di far operare il sistema sempre in modalità

Supervisor e che non è stata implementata alcuna *fast interrupt (FIQ)*, per ottimizzare gli spazi in memoria si è deciso di non allocare spazio per gli stack delle modalità non utilizzate.

La seconda parte dello script contiene informazioni su come si vuole che venga distribuito il codice oggetto all'interno dei blocchi di memoria. Nel caso in oggetto si è scelto di allocare il file di startup (file scritto in codice assembler) e il codice di programma nella memoria flash, mentre le variabili, ovviamente, nella RAM.

L'ultima parte, infine, contiene informazioni in formato *Stab* per la descrizione del programma al debugger.

Il file scritto in codice assembler è il file di startup vero e proprio. Nella prima parte, dopo la sezione di definizione delle costanti, viene scritta la *interrupt vector table*, ossia la tabella alla quale accede il controllore delle interruzioni (VIC) per individuare l'indirizzo della routine associata al tipo di interrupt verificatosi. Nella sezione successiva viene definita la modalità operativa e vengono disabilitate le interruzioni, così da poter richiamare il file di inizializzazione le cui operazioni, come vedremo, devono essere eseguite in maniera atomica per una corretto setup della scheda. Conclusa la fase di inizializzazione, vengono impostati gli indirizzi degli stack nei registri SP (Stack Pointer) per ogni modalità operativa utilizzata, vengono copiati i valori iniziali delle variabili della sezione DATA dalla memoria flash alla memoria RAM, vengono impostati a zero i byte della sezione BSS (sezione che contiene le variabili non inizializzate e lo spazio di memorizzazione comune) e infine, dopo aver riabilitato le interruzioni, viene richiamata la funzione *main*.

Il file di inizializzazione contiene le operazioni necessarie alla configurazione del microcontrollore e delle periferiche utilizzate. Si riporta di seguito il codice della funzione di inizializzazione:

```
void Initialize(void)
{
  MEMMAP = 0x1;
  //Configura il PLL
  SCS &= ~0x10;              //Imposta il range a 1~15
MHz
  SCS |= 0x20;               //Attiva il main clock
  PLLCON &= ~0x02;           //Disconnette il PLL
  PLLFEED = 0xAA;
  PLLFEED = 0x55;
  PLLCON &= ~0x01;           //Disabilita il PLL
  PLLFEED = 0xAA;
  PLLFEED = 0x55;
  while((SCS & 0x40) == 0);  //Attende che il main clock
                             //sia a regime
  CLKSRCSEL = 0x1;           //Seleziona il main clock
                             //come sorgente del PLL
  PLLCFG = 0x0B;             //M = 12, N = 1
  PLLFEED = 0xAA;
  PLLFEED = 0x55;
  PLLCON = 0x1;              //Abilita il PLL
  PLLFEED = 0xAA;
  PLLFEED = 0x55;
   while((PLLSTAT & (1<<26)) == 0); //Attende finché non
                                    //entra in lock
  while((PLLSTAT & 0xFF7FFF) != 0xB);//Attende che M e N
                                     //siano corretti
  USBCLKCFG = 0x5;           //USBClkSel=5
  CCLKCFG = 0x3;             //CClkSel=3 (72 MHz)
  while((PLLSTAT & (1<<26)) == 0);
  PLLCON |= 0x2;             //Connette il PLL
  PLLFEED = 0xAA;
  PLLFEED = 0x55;
  while((PLLSTAT & (1<<25)) == 0);
  //Configura il MAM
  MAMCR = 0x0;
  MAMTIM = 0x4;
  MAMCR = 0x2;
```

```
  // Configura i GPIO
  GPIOResetInit();
  PCONP = (EN_USB | EN_ETH | EN_UART1 | EN_TIM0);
  HC_CMD_STAT = 0x1;
}
```

La prima istruzione imposta la modalità di mappatura della memoria a *Flash Mode*: in questa modalità il vettore delle interruzioni non viene rimappato e risiede nella memoria Flash. L'utilizzo di questa modalità è dovuto principalmente alla decisione di far risiedere il codice nella memoria Flash, e quindi non si è ritenuto utile rimappare il vettore degli interrupt nella memoria RAM.

Le due istruzioni successive configurano e attivano il *main oscillator*, che verrà poi utilizzato come oscillatore in ingresso al PLL per generare le frequenze di clock necessarie.

Dopo l'attivazione dell'oscillatore viene eseguito il protocollo di configurazione e attivazione del PLL. Nel LPC2378 il PLL produce in uscita due frequenze distinte: una per la CPU e una per l'USB. I moltiplicatori e divisori del PLL sono stati scelti di modo da generare una frequenza di 48 MHz per la periferica USB e 72 MHz per la CPU; quest'ultima scelta è stata fatta poiché 72 MHz è la frequenza massima di funzionamento della CPU.

Al termine della configurazione del PLL vengono eseguite le istruzioni per la configurazione del *Memory Acceleration Module (MAM)*. Queste impostano il MAM per lavorare al massimo delle prestazioni e impostano il tempo di accesso alla memoria Flash a quattro cicli di clock.

La funzione *GPIOResetInit* inizializza le porte di I/O General Purpose, resettando tutti i pin GPIO ai valori di default.

L'istruzione successiva attiva la sorgente di clock per le periferiche utilizzate dal sistema: questa funzione è utile per il risparmio

energetico, dato che permette di attivare il funzionamento delle sole periferiche di interesse.
L'ultima istruzione è necessaria per risolvere un problema relativo al funzionamento della periferica USB (per ulteriori dettagli si rimanda agli Errata Sheet relativi al Philips LPC2378).

3.4.3 Il ModBus

In questo paragrafo verrà descritta l'architettura del protocollo ModBus, mettendo in luce le scelte implementative relative al microcontrollore LPC2378.

3.4.3.1 Caratteristiche generali

Il ModBus è un protocollo a scambio di messaggi, collocabile nel livello applicazione della pila OSI, che offre una comunicazione di tipo client/server fra dispositivi connessi con differenti tipi di bus e reti.
Lo stato dell'arte offre implementazioni su:

- TCP/IP over Ethernet;
- Trasmissioni seriali asincrone su una vasta tipologia di media (EIA/TIA-232-E, EIA-422, EIA/TIA-485-A, fibra, radio, ecc.);
- MODBUS PLUS, una rete a passaggio di testimone ad alta velocità.

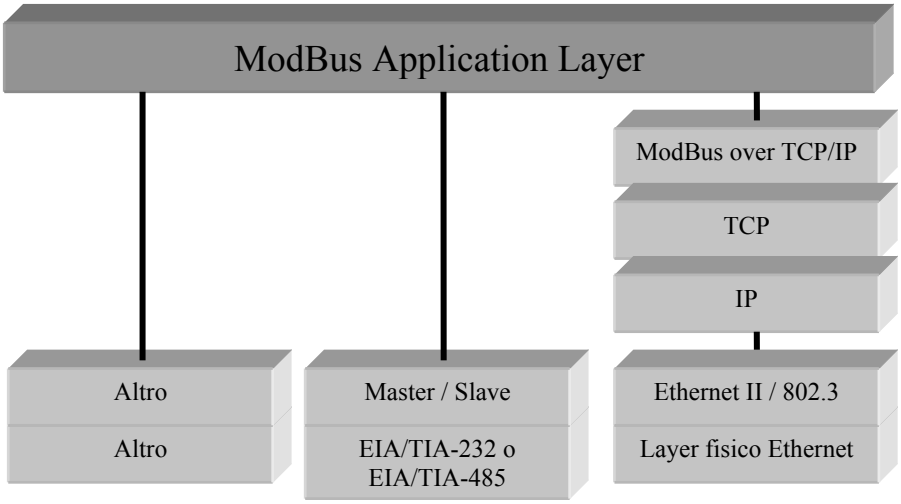

Figura 14: Stack di comunicazione ModBus

Il protocollo è di tipo richiesta/risposta e fornisce i servizi previsti per mezzo di **codici funzione**, trasmessi come elementi del PDU (protocol data unit) ModBus.

Tipi di architetture

Il protocollo ModBus consente la comunicazione anche fra diverse tipologie di rete e bus mediante l'utilizzo di gateway. Nella figura seguente si mostra un esempio di architettura di rete basata su questo protocollo.

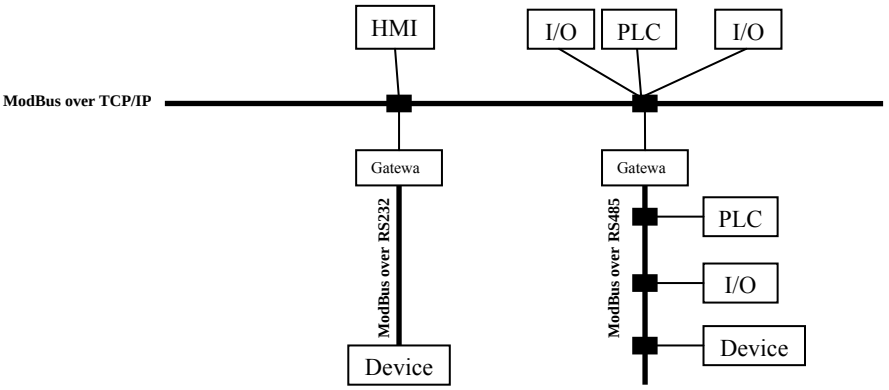

Figura 15: Esempio di architettura di rete ModBus

Ogni tipologia di dispositivo (PLC, Driver, HMI, ecc.) può utilizzare il protocollo ModBus per richiedere un'operazione su dispositivi remoti. Nel sistema ELIK, per il sottosistema Ethernet, è previsto l'utilizzo del protocollo proprietario AmicLAN: nonostante ciò, la caratteristica di indipendenza del ModBus dal mezzo trasmissivo non esclude la possibilità concreta di un futuro affiancamento del protocollo ModBus over Ethernet ad AmicLAN.

L'implementazione del protocollo si è basata sullo schema previsto per l'EIA/TIA-RS232 che è, per sua natura, un protocollo fisico di tipo punto-punto; nell'implementare il protocollo si è comunque tenuto conto della possibilità di un accesso master-multislave, così da permettere una facile migrazione all'EIA/TIA-RS485. Infatti, poichè i due protocolli fisici sono sviluppati in hardware, è sufficiente applicare un adattatore da un'interfaccia all'altra per il passaggio da un'implementazione all'altra.

La struttura del firmware per il protocollo ModBus è stata divisa in due moduli, nel rispetto dell'architettura dello stesso.

Il modulo *DLslave* si occupa delle funzioni di livello Data Link: esso gestisce i trasferimenti via seriale e, una volta completata la ricezione del *PDU serial link ModBus*, effettua il controllo degli indirizzi e il controllo dell'errore. Terminate queste operazioni, estrae il *PDU ModBus* dal PDU di livello Data Link, demandando la sua elaborazione allo strato superiore.

Il modulo *ALslave* implementa invece le funzioni di livello Application: il suo compito è estrarre il codice funzione dal messaggio, una volta ricevuto il *PDU ModBus*, ed eseguire la funzione appropriata.

Nei paragrafi seguenti verranno descritte nei particolari le operazioni eseguite da questi due moduli.

3.4.3.2 ModBus Data Link Protocol

Il protocollo ModBus su linea seriale è un protocollo di tipo Master-Slave che si localizza al livello 2 della pila OSI. Un sistema di questo tipo ha un nodo (il nodo master) che emette richieste di comandi espliciti ai nodi slave e elabora le risposte ottenute. I nodi slave, tipicamente, non trasmettono dati senza un'esplicita richiesta da parte del nodo master e non comunicano con gli altri nodi slave. A livello fisico il ModBus over Serial Line può usare due tipi di interfacce seriali:

- **Interfaccia RS232:** usata per brevi trasmissioni punto-punto;
- **Interfaccia RS485:** usata per trasmissioni punto-multipunto; la più usata è la Two-Wire, sebbene sia possibile implementare anche interfacce Four-Wire.

Il protocollo sviluppato per il sistema ELIK rientra nella categoria delle implementazioni **conditionally compliant**: essa implementa infatti solo i requisiti *mandatory* previsti dalle specifiche del protocollo. Questa scelta è stata influenzata dal fatto che i requisiti richiesti dalle specifiche sono, in realtà, stringenti solo nelle implementazioni del master dato che, se il master non implementa alcune funzioni, potrebbe non essere in grado di comunicare con alcuni slave, mentre lo slave è libero di non implementare alcuni requisiti ritenuti non necessari senza compromettere il corretto funzionamento dell'intero sistema. Per questioni di risparmio di memoria fisica si è deciso, quindi, di implementare solo i requisiti strettamente necessari, non precludendosi, ad ogni modo e se necessario, la possibilità di aggiornare il sistema con funzioni aggiuntive.

Principi del protocollo Master-Slave ModBus

Il protocollo ModBus over Serial Line prevede la connessione di un solo master per bus, mentre possono essere presenti fino a un massimo di 247 nodi slave. Una comunicazione ModBus può essere iniziata solo dal master: i nodi slave non possono trasmettere dati senza ricevere un'esplicita richiesta dal nodo master e non possono comunicare fra loro. Un nodo master può iniziare una sola richiesta alla volta e può effettuare una richiesta ai nodi slave in due modi:

- **Modo Unicast:** il master indirizza un singolo slave. Dopo aver ricevuto e processato la richiesta lo slave restituisce al master un messaggio di risposta;

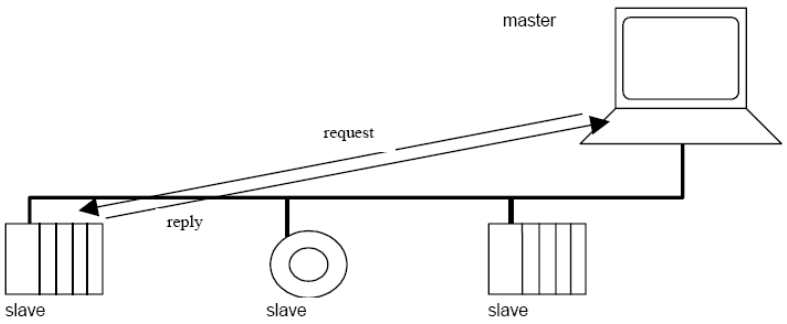

Figura 16: Modalità di comunicazione unicast

- **Modo Broadcast:** il master invia una richiesta a tutti i nodi slave. Gli slave, in questo caso, non gli restituiscono alcuna risposta. La richiesta broadcast è necessariamente un comando di scrittura e tutti i dispositivi devono accettare questo tipo di richieste. L'indirizzo 0 è riservato alla loro identificazione.

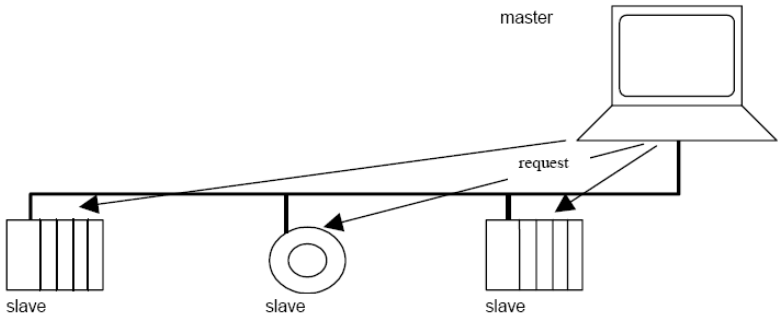

Figura 17: Modalità di comunicazione broadcast

Regole di indirizzamento

Lo spazio di indirizzamento ModBus comprende 256 indirizzi differenti:

0	Da 1 a 247	Da 248 a 255
Indirizzo broadcast	Indirizzo univoco Slave	Riservati

Figura 18: Ripartizione dello spazio degli indirizzi ModBus

L'indirizzo 0 è riservato come indirizzo broadcast e tutti i nodi slave devono essere in grado di riconoscerlo.
Il nodo master non ha un indirizzo specifico, mentre gli slave possono avere un indirizzo, che deve essere unico in un bus seriale.

Descrizione del frame

Al PDU di livello applicazione del ModBus si introducono campi aggiuntivi per la trasmissione su linea seriale. Il client che inizia una transazione costruisce il PDU ModBus e lo impacchetta nel PDU serial link ModBus, che ha i seguenti campi:

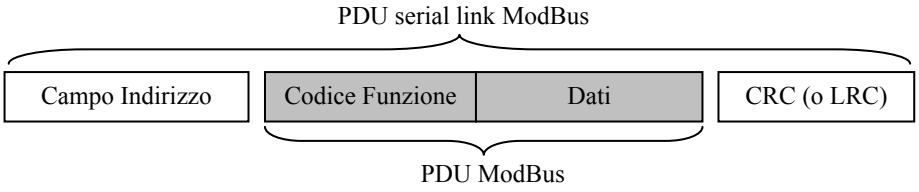

Figura 19: PDU serial link ModBus

- Il campo indirizzo contiene l'indirizzo dello slave: quando uno slave restituisce una risposta, scrive il suo indirizzo nel *campo indirizzo* di risposta, cosicché il master possa sapere da quale slave è arrivata;
- Il campo di controllo dell'errore è il risultato di un calcolo di controllo di ridondanza eseguito sul PDU.

Quando è terminata la ricezione di un PDU serial link ModBus, il modulo *DLslave* controlla innanzitutto l'indirizzo del messaggio: se l'indirizzo contenuto nel *campo indirizzo* corrisponde o al proprio indirizzo slave, o all'indirizzo broadcast, verrà verificata la correttezza del CRC, richiamando la funzione *Mb_test_crc*. Se anche quest'ultimo controllo restituisce un esito positivo (0 se corretto, 1 se scorretto), il PDU verrà trasmesso al layer successivo. Terminate le operazioni del *ALslave*, se il *campo indirizzo* conteneva l'indirizzo broadcast non viene trasmesso nulla, altrimenti verrà trasmesso il messaggio di risposta ricevuto dal layer superiore. Per fare ciò, il modulo calcola il CRC del PDU ModBus ricevuto dal layer superiore, richiamando la funzione *Mb_calcul_crc*, appende l'indirizzo dello slave al PDU e trasmette il PDU così costruito all'interfaccia seriale.
Si riporta, per una maggiore comprensione, il frammento di codice che implementa le operazioni suddette:

```
[ … ]
While(!Mbs_read(trame,&length));
if((trame[0] == MB_ADDRESS_BROADCAST || trame[0] ==
            Mbs_slave) && !Mb_test_crc(trame,length-2)){
  Mb_ALslave(PDU,&PDUresLength);
  if(!(trame[0]==MB_ADDRESS_BROADCAST)){
    Mb_calcul_crc(trame,PDUresLength+1);
    Mbs_write(trame,PDUresLength+3);
  }
}
[ … ]
```

Le funzioni *Mbs_read* e *Mbs_write* leggono e scrivono sull'interfaccia seriale, seguendo le specifiche del protocollo.

Diagramma a stati finiti dello slave

Lo strato Data Link del ModBus comprende due sottostrati separati:

- Il protocollo master/slave;
- Il transmission mode:
  - RTU
  - ASCII

Di seguito si riporta la parte del diagramma a stati finiti dello slave, indipendente dal transmission mode. Non verrà riportato invece la parte relativa al master, poiché non è oggetto di analisi per lo sviluppo del sistema ELIK.

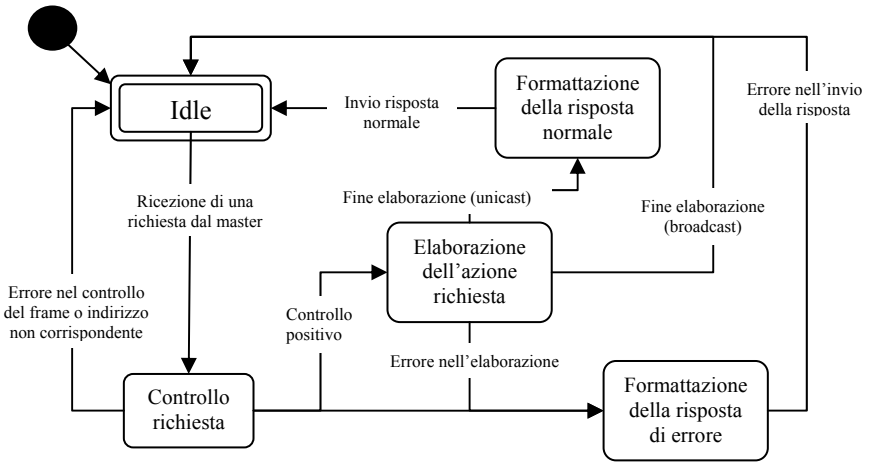

Figura 20: Diagramma a stati finiti dello slave ModBus

Diagramma temporale di comunicazione

Qui sotto si mostra il diagramma temporale di tre tipici scenari di comunicazione.

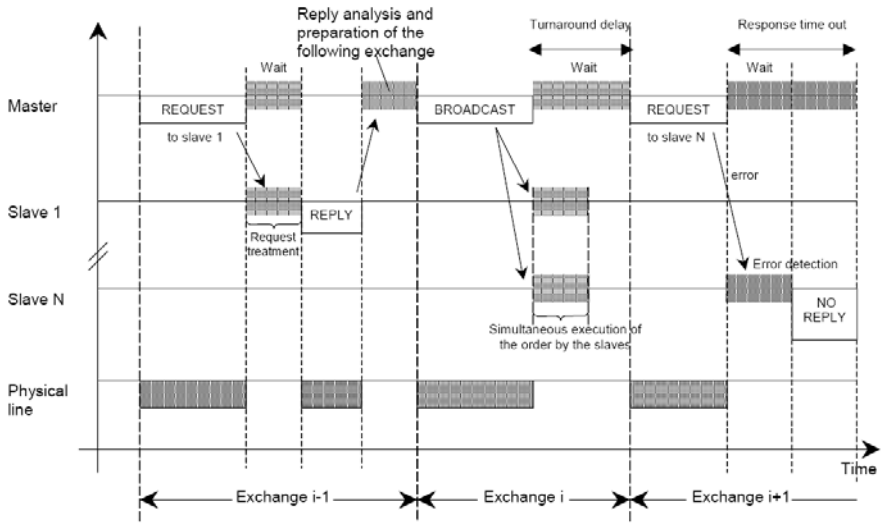

Figura 21: Diagramma temporale di tre possibili scenari ModBus

E' da sottolineare che:

- La durata delle fasi REQUEST, REPLY e BROADCAST dipende dalle caratteristiche di comunicazione (lunghezza del frame, throughput);
- La durata delle fasi WAIT e TREATMENT dipende dal tempo di elaborazione della richiesta per l'applicazione slave.

I due modi di trasmissione

Nel protocollo sono definiti due modi di trasmissione: il RTU Mode e l'ASCII Mode.

Questi definiscono i bit contenuti nei campi del messaggio trasmessi serialmente nella linea, e determinano quindi la modalità di raggruppamento di questi campi e le regole di decodifica dell'informazione ivi contenuta. Ovviamente, il transmission mode e i parametri della porta seriale devono essere gli stessi per tutti i dispositivi presenti nella linea seriale.

Di seguito verrà descritto solo il RTU Mode, giacché è il transmission mode implementato nel sistema ELIK; dal momento che l'ASCII Mode è opzionale, si è scelto di trascurarne l'implementazione per diminuire la complessità dello slave.

RTU Transmission Mode

Quando un dispositivo comunica in una linea seriale usando la modalità RTU (Remote Terminal Unit), ogni byte di 8-bit nel messaggio contiene due caratteri esadecimali. Il principale vantaggio di questa modalità è che la sua maggiore densità consente, a parità di baudrate, un throughput più elevato rispetto alla modalità ASCII, che invece, per ogni byte di 8-bit, trasmette un carattere ASCII. Ogni messaggio deve essere trasmesso in un flusso continuo di caratteri.

Il frame di livello fisico che permette la trasmissione di un byte di dati prevede un bit di start, un bit di stop e un bit di parità, o un altro bit di stop nel caso in cui non si reputi necessario il controllo di parità.
Nella figura seguente si mostra la struttura del frame di livello Data Link per la modalità di trasmissione RTU, che prevede un campo indirizzo, il PDU Modbus e un campo di controllo di ridondanza ciclico.

Indirizzo Slave	Codice Funzione	Dati	CRC
1 byte	1 byte	Da 0 a 252 byte	2 byte

Figura 22: Numero di byte dei campi del frame RTU

Frammentazione del messaggio RTU

Un messaggio ModBus è impacchettato dalla periferica di trasmissione in un frame che ha un punto di inizio e uno di fine conosciuto. Ciò consente al dispositivo che riceve un nuovo frame di riconoscere l'inizio del messaggio e di sapere quando la ricezione è completata. E' necessario rilevare messaggi parziali e questa evenienza va considerata come errore.
In questa modalità, i frame sono separati da un intervallo di silenzio di almeno 3.5 character times.

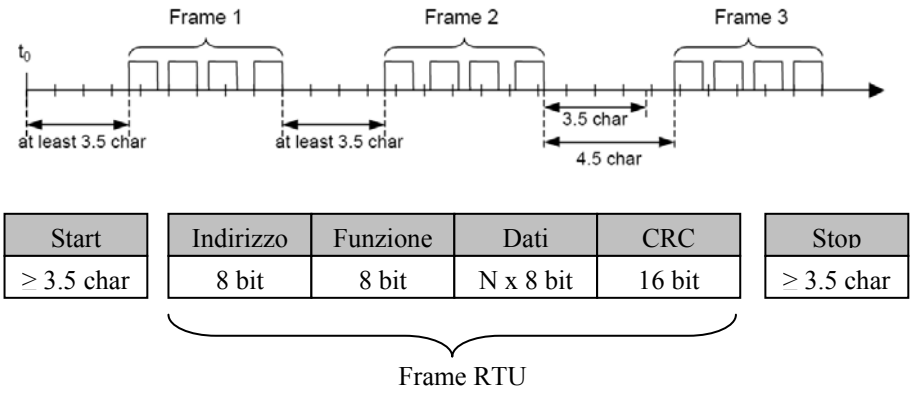

Figura 23: Vincoli di tempo per il riconoscimento di un frame RTU

Dato che l'intero messaggio deve essere trasmesso in un flusso continuo di caratteri, se viene prodotto un intervallo di silenzio di più di 1.5 character times fra un carattere e l'altro, il frame del messaggio viene dichiarato incompleto e deve essere scartato dal ricevente.
La seguente figura mostra una descrizione del diagramma a stati di una trasmissione in modalità RTU. Nello stesso grafico appaiono sia il master che lo slave.

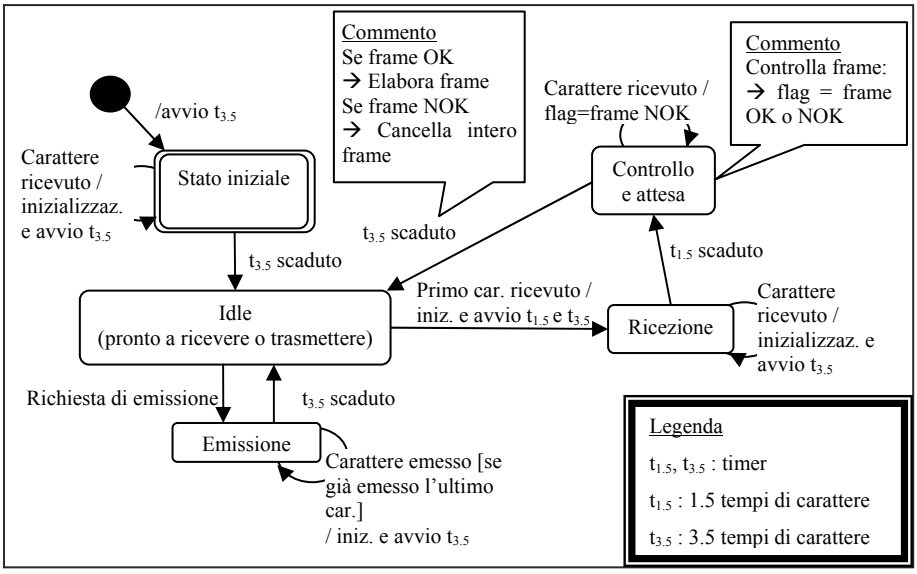

Figura 24: Diagramma a stati finiti di una trasmissione in modalità RTU

Dopo la fase di inizializzazione, che configura il timer e l'interfaccia UART secondo le specifiche dell'LPC2378, viene richiamato il modulo *DLslave* il quale, dopo aver verificato la coerenza delle impostazioni delle variabili interne, si pone in attesa di ricezione di un carattere dall'interfaccia seriale, richiamando la funzione *Mbs_read*. Quando si riceve il primo carattere, viene richiamata la funzione *timer1_en* che resetta il contatore e avvia il timer. Durante questa fase, il sistema rimane in attesa dello scadere del timeout $t_{1,5}$ che indica la completa ricezione del PDU; per il controllo del valore del timer viene utilizzata la funzione *timer1_Chk*. A questo punto viene settato il flag *fine_trama* e si attende lo scadere del timeout $t_{3,5}$. Se prima dello

scadere di questo l'interfaccia riceve un nuovo carattere, la trasmissione viene considerata incompleta e il PDU viene scartato. Ogniqualvolta viene ricevuto un nuovo carattere dalla periferica UART esso viene memorizzato nel buffer *UART1Buffer*, viene incrementata la variabile contatore *UART1Count*, nella quale viene memorizzato il numero di caratteri ricevuti, e il timer viene resettato richiamando la funzione *timer1_en*.
Si riporta di seguito il frammento di codice riguardante la gestione della UART:

```
[ … ]
int Mb_read(BYTE *BufferPtr, DWORD* Length )
{
  BYTE i;
  UART1Count = 0;
  while(UART1Count==0);
  Timer1_En();
  while(Timer1_Chk() < TIMEOUT15);
  *Length=UART1Count;
  Wait = 1;
  while(Timer1_Chk() < TIMEOUT35);
  if(UART1Count != *Length) return 0;
      UART1Count = 0;
  Wait = 0;
  for(i=0;i<*Length;i++)
    BufferPtr[i] = UART1Buffer[i];
  return 1;
}
[ … ]
```

La variabile globale *Wait* viene utilizzata per indicare alla routine di interrupt di non resettare il timer, nel caso in cui ci si trovi nello stato di *Controllo e attesa*.

Il calcolo dei parametri TIMEOUT15 e TIMEOUT35 viene fatto sulla base del baud rate scelto. Tenendo conto che la granularità del timer è fissata al decimo di microsecondo, la formula per il calcolo dei parametri è la seguente:

$$TIMEOUT15 = \left| 1{,}5 \times \frac{8 \times 10000}{BaudRate} \right| \mu s$$

$$TIMEOUT35 = \left| 3{,}5 \times \frac{8 \times 10000}{BaudRate} \right| \mu s$$

Controllo CRC

La modalità RTU include un campo di controllo dell'errore basato sul metodo del controllo di ridondanza ciclico (CRC), eseguito sul contenuto del messaggio.

Il campo CRC racchiude informazioni sull'intero contenuto del messaggio, e si applica indipendentemente dal metodo di controllo di parità usato.

Il campo CRC contiene un valore di 16 bit implementato come 2 bytes da 8 bit. Questo viene "appeso" al messaggio come ultimo campo. Per fare ciò, il byte del campo di ordine più basso viene appeso per primo, seguito da quello di ordine più alto; quest'ultimo è l'ultimo byte trasmesso nel messaggio.

Il valore di CRC viene calcolato dal dispositivo mittente; il dispositivo destinatario ricalcola il CRC durante la ricezione del messaggio e lo confronta con il valore effettivo ricevuto nel campo CRC: se i due valori coincidono il frame è stato ricevuto correttamente.

Il calcolo del CRC inizia caricando un registro di 16 bit con tutti i bit a 1. A questo punto comincia l'elaborazione, applicando i bytes di 8 bit successivi del messaggio al contenuto corrente del registro. Per generare il CRC sono utilizzati solo gli 8 bit di dati in ogni carattere.

Durante la generazione del CRC, per ogni carattere di 8 bit viene calcolato un OR esclusivo fra questo e il contenuto del registro, quindi il risultato viene shiftato a destra. Il LSB viene estratto e esaminato: se questo vale 1 si effettua uno XOR fra il valore contenuto nel registro e un valore prefissato, altrimenti non viene fatto nulla.
Questo processo viene ripetuto finché non vengono eseguiti otto shift. Dopo l'ultimo shift, il successivo byte di 8 bit viene combinato mediante uno XOR con il valore corrente del registro e il processo si ripete, come sopra, per altri otto shift. Il contenuto finale del registro è il valore del CRC.
Nell'implementazione del protocollo la gestione del controllo di ridondanza ciclico viene effettuata per mezzo delle due funzioni *Mb_test_crc* e *Mb_calcul_crc*.
La prima viene utilizzata durante la fase di lettura del modulo *DLslave*. Esso riceve in ingresso il PDU e la lunghezza del pacchetto senza il campo CRC e, da queste informazioni, calcola il CRC del PDU ricevuto e lo confronta con il CRC ricevuto: se i due CRC corrispondono restituisce 0, altrimenti 1.
La seconda funzione, invece, viene utilizzata nella fase di scrittura del modulo *DLslave* per calcolare il CRC da appendere al *PDU serial link ModBus*, che verrà successivamente trasmesso via interfaccia seriale.

3.4.3.3 ModBus Application Protocol

Il protocollo ModBus definisce un semplice PDU indipendente dallo strato di comunicazione sottostante. Il mapping del protocollo su alcuni tipi di bus o reti può introdurre alcuni campi opzionali all'ADU (application data unit).

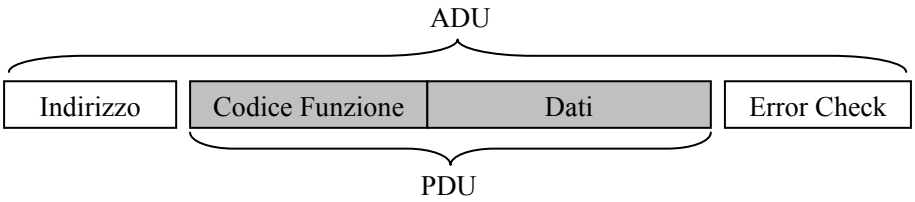

Figura 25: Formato dell'ADU dell'application layer ModBus

L'ADU ModBus è costruito dal client che inizia la transazione. Il *Codice Funzione* indica al server quale operazione deve compiere e il campo *Dati* contiene informazioni aggiuntive, utili al server per compiere l'operazione richiesta.

Il codice funzione è codificato su otto bit, ma di questi un bit è riservato per identificare il verificarsi di un'eccezione. Quindi, il protocollo può implementare al più 127 funzioni diverse, poiché il codice funzione 0 non è utilizzabile.

Alcune funzioni possono prevedere un codice funzione aggiuntivo, detto *codice di sottofunzione*, per definire un set di funzioni più vasto.

Il campo dati può includere indirizzi di registri, quantità di elementi da gestire e, nel caso in cui la dimensione di questo campo non sia fissa, anche la dimensione in bytes. In alcuni tipi di richieste questo campo può anche essere inesistente.

Se nella richiesta non si verificano errori, il campo dati della risposta contiene i dati richiesti, mentre il campo codice funzione contiene una replica del codice funzione ricevuto.

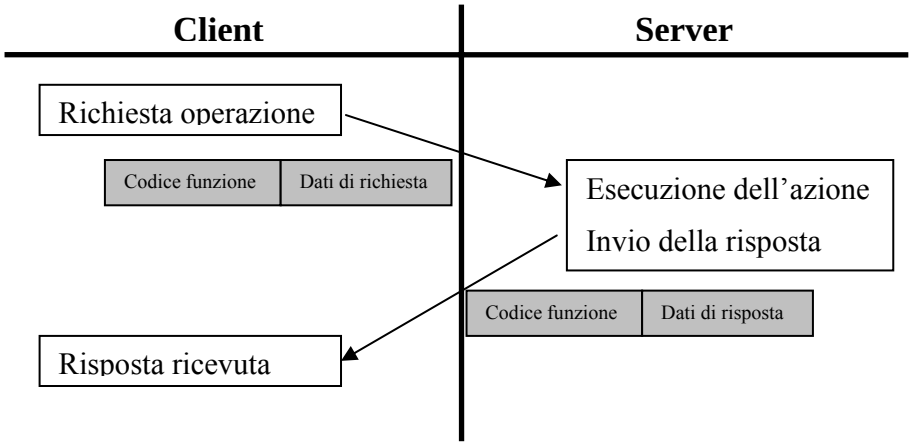

Figura 26: Esempio di handshake senza errore

Se invece si verifica un errore, il campo dati contiene un codice di eccezione che l'applicazione client potrà utilizzare per determinare l'operazione successiva da eseguire, mentre il campo codice funzione è lo stesso della richiesta, ma con il bit di eccezione settato.

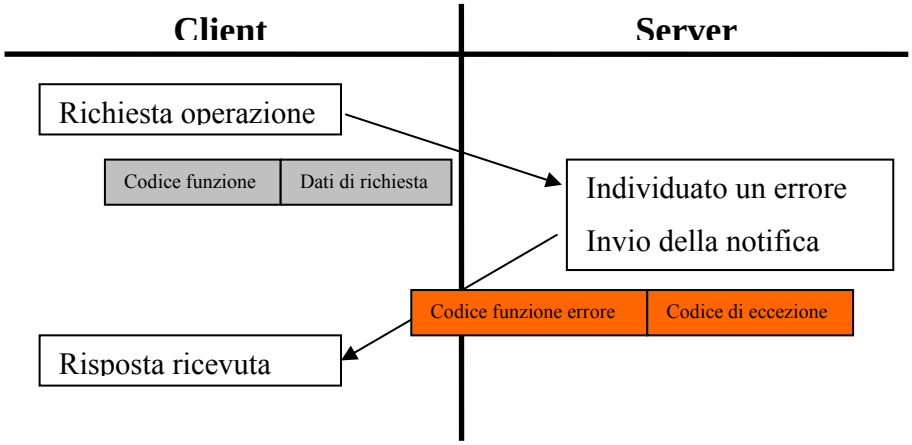

Figura 27: Esempio di handshake con errore

La dimensione del PDU è limitata dai vincoli di dimensione della prima implementazione del ModBus sviluppata, ossia quella su linea seriale (RS485 ADU:256 bytes).

Quindi il PDU per comunicazioni seriali ha dimensioni massime pari a:

PDUmax = 256 B – 1 B (Indirizzo server) – 2 B (CRC) = **253 Bytes**

Dalla descrizione fatta si può concludere che il protocollo ModBus definisce tre tipi di PDU:

• MODBUS Request PDU;

• MODBUS Response PDU;

• MODBUS Exception Response PDU.

Modello dei dati

ModBus usa la rappresentazione *big-endian* per gli indirizzi e i dati: questo significa che, quando trasmette una quantità numerica maggiore di un byte, il primo byte viene inviato per primo.

Il protocollo basa il suo modello dei dati su una serie di tabelle con differenti caratteristiche. Le quattro tabelle primarie sono:

Tabella 2: Tabelle primarie relative al modello dei dati ModBus

Tabelle primarie	Tipo di oggetto	Modalità di accesso	Commenti
Discrete input	Singolo bit	Sola lettura	Offerti da un sistema di I/O
Coil	Singolo bit	Lettura/Scrittura	Alterabili da un'applicazione
Input register	Word a 16 bit	Sola lettura	Offerti da un sistema di I/O
Holding Register	Word a 16 bit	Lettura/Scrittura	Alterabili da un'applicazione

La distinzione fra dati di ingresso e di uscita e fra dati indirizzabili al bit e al word non influisce sul comportamento dell'applicazione. E' prevista, infatti, la possibilità che le quattro tabelle siano sovrapposte, se si ritiene che questo sia il modo migliore per interpretare il contenuto informativo della macchina target.

Per ognuna delle tabelle primarie il protocollo consente la selezione di 65536 elementi, e le operazioni di lettura e scrittura sono progettate

per essere eseguite su più elementi consecutivi, fino al limite di dimensione che dipende dalla funzione richiesta.
I dati gestiti dall'applicazione ModBus vengono mappati nella memoria fisica: non vanno quindi confusi gli indirizzi che individuano il dato a livello applicazione con gli indirizzi fisici in cui risiedono effettivamente i dati. I riferimenti logici del ModBus sono interi senza segno che partono da zero.

Uno dei problemi di maggior rilievo riscontrati nello sviluppo delle funzioni di protocollo ModBus è stato l'accesso al bit; infatti, la granularità minima di accesso alla memoria per il linguaggio C è il byte. Si è resa perciò necessaria la formulazione di un algoritmo che, sfruttando gli operatori binari, permettesse l'accesso in memoria al bit. Per fare ciò sono state sviluppate due funzioni che consentono la lettura e la scrittura di bit di dati: *GetBits* e *SetBits*.
La funzione *GetBits* riceve in ingresso il buffer da cui estrarre il bit e l'indirizzo del bit di cui si vuole conoscere il valore. Il codice che implementa questa funzionalità è il seguente:

```
[ … ]
usByteOffset = ( WORD )( ( OffsetBitInteresse ) / 8 );
usNPreBits = ( WORD )( OffsetBitInteresse –
                       OffsetByteInteresse * 8 );
usMask = 1 << usNPreBits;
ValoreBit = (Buffer[OffsetByteInteresse] &
                       usMask) >> usNPreBits;
[ … ]
```

L'algoritmo calcola, prima di tutto, l'indirizzo effettivo del byte in memoria che contiene il bit d'interesse, dividendo l'indirizzo del bit per 8; calcola poi il numero di bit che precedono il bit che si vuole estrarre e, noto questo valore, ricava la maschera che servirà ad

estrarre il bit richiesto (si ricorda che il ModBus utilizza, per gli indirizzi e i dati, la rappresentazione big-endian).
La funzione *SetBits* riceve in ingresso il buffer su cui dovrà modificare il bit e l'indirizzo del bit che si vuole manipolare. Il codice che implementa questa funzionalità è il seguente:

```
[ … ]
usByteOffset = ( WORD )( (OffsetBitInteresse) / 8 );
usNPreBits = ( WORD )( OffsetBitInteresse –
                              OffsetByteInteresse * 8 );
ucValue <<= usNPreBits;
usMask = ~( 1 << usNPreBits );
usBuf = Buffer[OffsetByteInteresse] & usMask;
usBuf |= ucValue;
Buffer[OffsetByteInteresse] = usBuf;
[ … ]
```

Come il precedente, l'algoritmo calcola, innanzitutto, l'indirizzo effettivo del byte in memoria che contiene il bit d'interesse, dividendo l'indirizzo del bit per 8; ricava, a questo punto, il numero di bit che precedono il bit che si vuole manipolare e, noto questo valore, calcola la maschera che servirà a modificare, del byte che lo contiene, solo il bit d'interesse con una serie di operazioni bitwise.

Per comprendere meglio come vengono mappati i dati ModBus nella memoria fisica, si mostreranno ora due possibili modelli di mappatura.

Dispositivo con quattro blocchi separati

In Figura 28 è mostrato un esempio in cui i diversi tipi di dati sono mappati in memoria in locazioni differenti, poiché i dati non sono correlati fra loro:

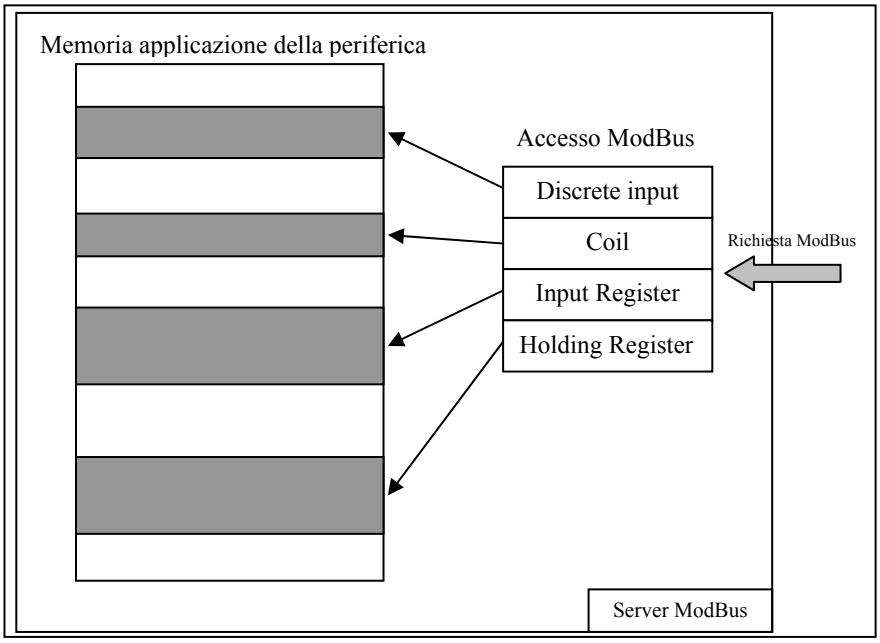

Figura 28: Esempio di mappatura della memoria come quattro blocchi separati

Dispositivo con un solo blocco

In Figura 29 è invece mostrato un esempio in cui i diversi tipi di dati sono mappati in memoria nello stesso blocco:

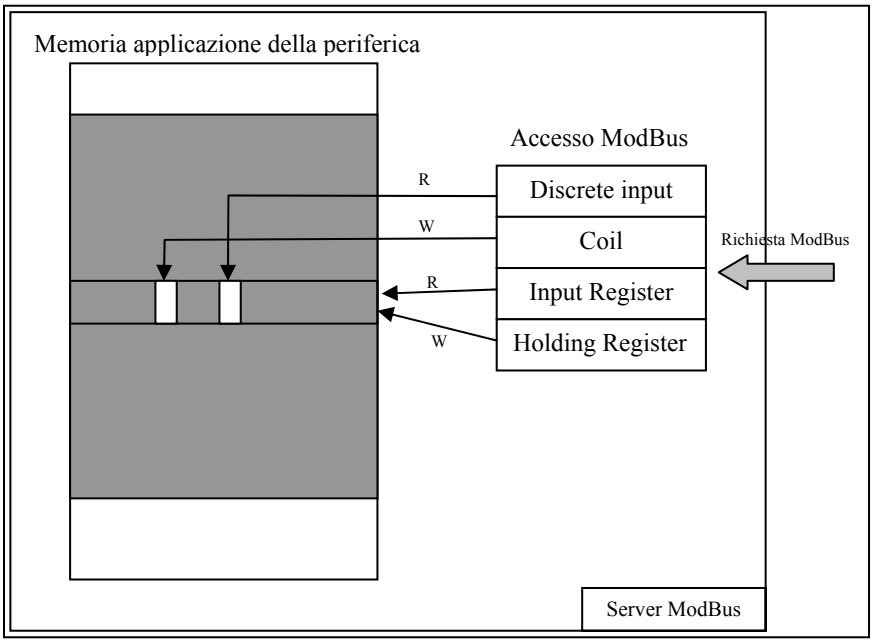

Figura 29: Esempio di mappatura della memoria come unico blocco

Modello di indirizzamento

Il protocollo definisce regole di indirizzamento ben precise:

1. In un PDU ogni dato è indirizzato con valori da 0 a 65535;
2. Nel modello di dati ogni elemento all'interno di un blocco dati è numerato da 1 a n;
3. La mappatura fra il modello dei dati e la memoria del dispositivo dipende totalmente dalla scelta dello sviluppatore del dispositivo stesso.

In Figura 30 è mostrato un esempio di accesso alla memoria logica e rispettivo reindirizzamento alla memoria fisica.

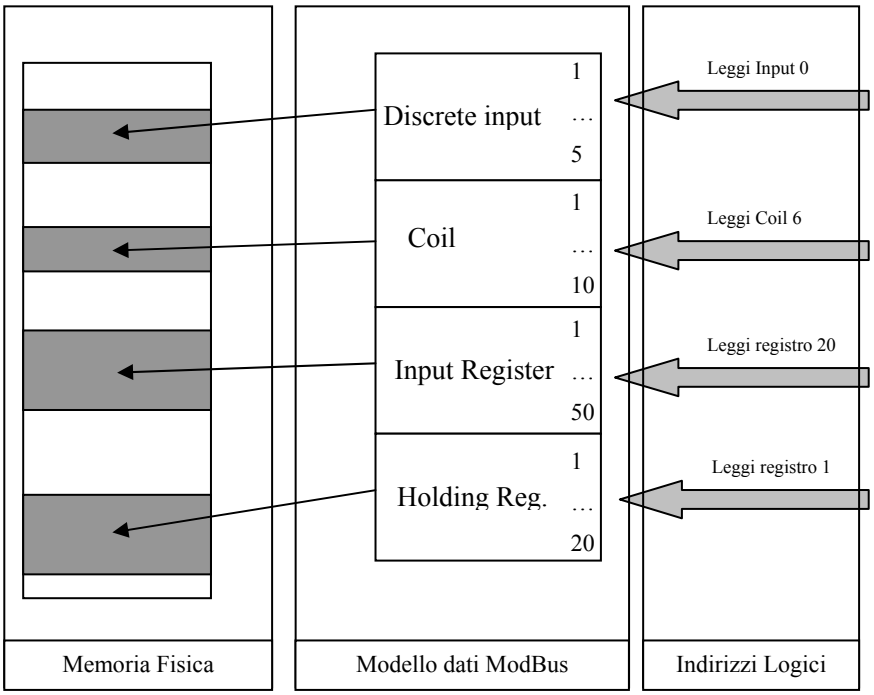

Figura 30: Modello di indirizzamento ModBus

Le transazioni ModBus

Il seguente diagramma a stati finiti descrive la procedura generale seguita dal lato server durante una transazione:

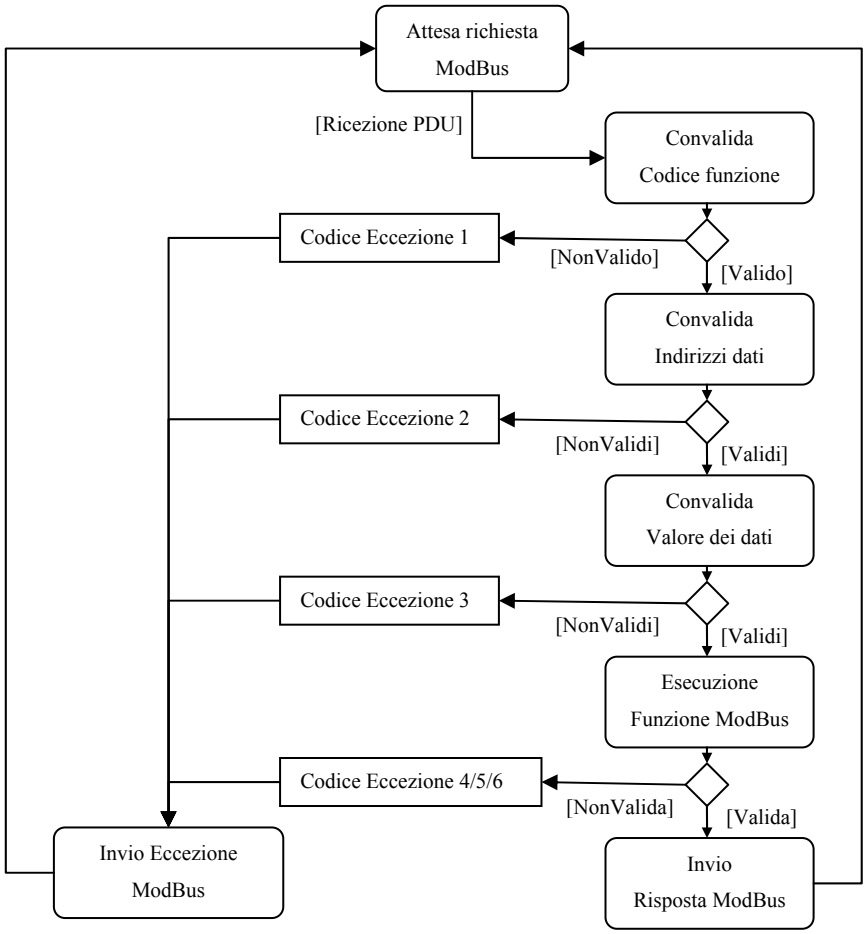

Figura 31: Diagramma a stati finiti per una transazione ModBus lato slave

Appena una richiesta è stata elaborata dal server, viene prodotta la risposta in funzione del risultato ottenuto al termine dell'elaborazione stessa. Sono previsti dal protocollo due possibili risposte:

- Risposta positiva:
  - *Codice funzione della risposta = Codice funzione della richiesta*.
- Eccezione:
  - Fornisce al client informazioni rilevanti sull'errore prodotto durante l'elaborazione;
  - *Codice funzione dell'eccezione = Codice funzione della richiesta* + 0x80;

Tipi di codice funzione

I codici funzione possono essere raggruppati in tre categorie:

1. **Codici funzione pubblici:**
   Sono codici funzione well-known, unici e documentati;
2. **Codici funzione definiti dall'utente:**
   Sono codici funzione che possono essere utilizzati come codici *ad hoc* per un determinato dispositivo;
3. **Codici funzione riservati:**
   Sono codici funzione usati da alcune compagnie per determinati prodotti e per questo non disponibili al pubblico utilizzo.

Nel sistema ELIK sono state sviluppate solo alcune funzioni associate ai *Codici Funzione Pubblici*. La disponibilità di codici funzione definibili dall'utente ha reso ancora più valida la scelta del protocollo ModBus come standard per l'automazione industriale nel progetto in questione: questa possibilità, infatti, rende il protocollo molto flessibile e scalabile, due fattori assai rilevanti per un sistema pensato come possibile substrato di future applicazioni domotiche.

Descrizione dei codici funzione

Di seguito si riporta il sottoinsieme delle funzioni ModBus realizzate per il sistema ELIK:

- **Read Coils**: utilizzata per leggere lo stato di massimo 2000 coils contigui in un dispositivo remoto;
- **Read discrete input:** utilizzata per leggere lo stato di massimo 2000 ingressi discreti contigui in un dispositivo remoto;
- **Read Holding Registers:** usata per leggere il contenuto di blocchi contigui di holding registers in un dispositivo remoto;

- **Read Input Registers:** usata per leggere il contenuto di blocchi contigui di massimo circa 125 input registers in un dispositivo remoto;
- **Write single coil:** usata per settare un singolo coil a ON o a OFF in un dispositivo remoto;
- **Write single register:** usata per scrivere un solo holding register nel dispositivo remoto;
- **Read Exception status:** usata per leggere il contenuto di otto valori di stato delle eccezioni nel dispositivo remoto (il significato degli otto valori di stato delle eccezioni è dipendente dall'applicazione);
- **Diagnostics:** permette di eseguire una serie di test sul sistema di comunicazione fra il dispositivo client e quello server, o di controllare alcune condizioni di errore interni al server (di questa è stata implementata solo la sottofunzione ***Return Query Data***);
- **Write Multiple Coils:** usata per forzare ogni coil nella sequenza di coil a ON o a OFF in un dispositivo remoto;
- **Write Multiple Register:** usata per scrivere un blocco di registri contigui (da 1 a 120 registri) in un dispositivo remoto;
- **Report Slave ID:** usata per leggere la descrizione del tipo, dello stato corrente e altre informazioni specifiche di un dispositivo remoto;
- **Read/Write Multiple registers:** esegue nell'ordine una operazione di scrittura e una di lettura di holding registers in una singola transazione.

Il flow chart di Figura 31 è stato implementato nel modulo *ALslave* come istruzione switch sul campo *Codice Funzione* del PDU di richiesta ricevuto: in funzione del codice contenuto nel campo, viene

richiamata la funzione che esegue le operazioni ModBus appropriata. Il seguente frammento di codice esemplifica la struttura del modulo:

```
[ … ]
switch(PDUreq[0]){
  case FC_RCOILS:
    res=ReadCoils(PDUreq,PDUresLength);
    break;
  case FC_RDISCIN:
    res=ReadDInputs(PDUreq,PDUresLength);
    break;
     [ … ]
  default:
    sendException(PDUreq,PDUresLength,1);
}
if(res==2)
  sendException(PDUreq,PDUresLength,4);
[ … ]
```

Le funzioni *ReadCoils* e *ReadDInputs* sono due delle funzioni C che implementano le funzioni ModBus gestite, mentre la funzione *sendException* si occupa di inviare al master il PDU di errore appropriato per l'eccezione che è stata catturata.

Risposta alle eccezioni

Quando un dispositivo client invia una richiesta al dispositivo server, esso si aspetta una risposta positiva. In seguito ad una richiesta si può verificare uno dei seguenti quattro eventi:

- Se il dispositivo server riceve la richiesta senza errori di comunicazione e può gestirla normalmente, restituisce una risposta positiva;
- Se il server non riceve la richiesta in seguito ad un problema di comunicazione non viene restituita alcuna risposta. Il

programma client processerà eventualmente una condizione di timeout associato alla richiesta;

- Se il server riceve una richiesta ma individua un errore di comunicazione (parità, CRC, ecc.) non viene restituita alcuna risposta. Il programma client processerà eventualmente una condizione di timeout associato alla richiesta;
- Se il server riceve una richiesta senza errori di comunicazione, ma non può gestirla (per esempio se è richiesto di leggere un registro inesistente), questo restituisce un'eccezione informando il client sulla natura dell'errore.

Verranno elencati di seguito i tipi di eccezione gestiti dal sistema ELIK con i relativi significati:

- **Illegal Function:** il codice funzione ricevuto nella richiesta non è un'azione consentita al server; questa eccezione si verifica se, ad esempio, la funzione non è implementata nel server;
- **Illegal Data Address:** l'indirizzo dei dati ricevuto nella richiesta non è un indirizzo accessibile per il server. In particolare, la combinazione di indirizzo di partenza e lunghezza provoca un accesso a valori inesistenti;
- **Illegal Data Value:** il valore contenuto nel campo dati della richiesta non è un valore accettabile dal server;
- **Slave Device Failure:** si è verificato un errore irrecuperabile durante il processo di evasione della richiesta;

Nell'implementazione sviluppata le eccezioni *Illegal Function* e *Slave Device Failure* vengono gestite direttamente dal modulo *ALslave*, mentre le restanti vengono catturate direttamente dalla funzione, poiché dipendono dall'operazione richiesta dal master.

Come già anticipato precedentemente, l'invio del PDU di eccezione è demandato alla funzione *SendException*, che riceve in ingresso il PDU che ha generato l'eccezione, la sua lunghezza e il codice di eccezione associato all'errore verificatosi.

3.4.4 Il protocollo AmicLan

In questo paragrafo verranno descritte le caratteristiche generali del protocollo AmicLan; nella trattazione verranno omesse le specifiche in dettaglio e l'implementazione, poichè il protocollo è proprietario dell'Amic S.r.l. e quindi protetto da vincoli di riservatezza.

AmicLan è un protocollo per lo scambio di messaggi di tipo richiesta/risposta fra dispositivi. L'accesso è di tipo single master/multi slave, ed è pensato per lavorare su bus seriali di tipo half-duplex. I dispositivi attori possono essere ripartiti in tre classi:

- **M0 - classe *MASTER***: a questa classe appartengono i dispositivi destinati a collegarsi con i dispositivi slave solo tramite un dispositivo Master di classe 1;
- **M1 - classe *CONTROLLER***: a questa classe appartiene un dispositivo (che può anche non essere collegato alla rete) che ha il compito di monitoraggio, ed eventualmente anche di controllo, dell'intero sistema. Questo dispositivo può essere delegato a rendere disponibili le risorse del sistema ad utenti esterni (sia in locale, tramite una porta RS232, sia in remoto tramite un modem o altro);
- **M2 - classe *DEVICE***: fanno parte di questa classe tutti i generici dispositivi di acquisizione dati, attuazione di relé, uscite analogiche, e così via.

In Figura 32 si mostra un possibile scenario:

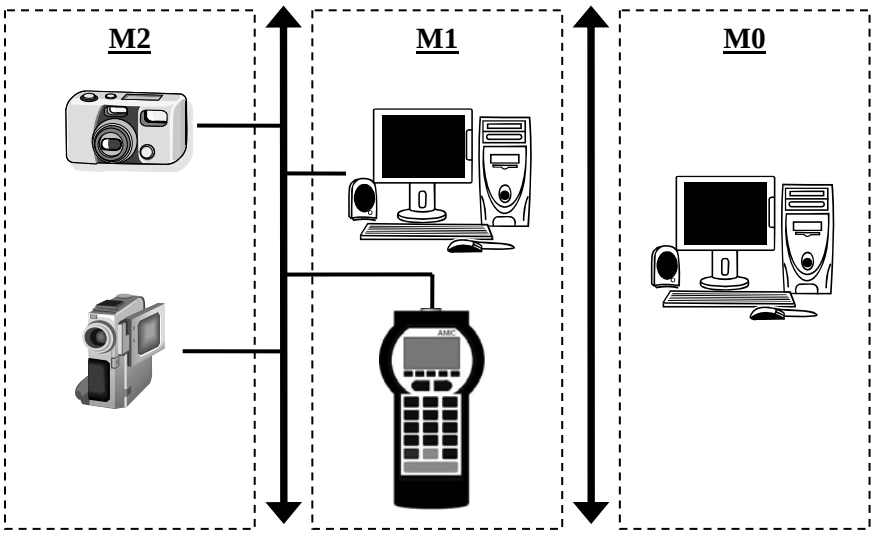

Figura 32: Possibile scenario per una rete AmicLan

Il PDU di richiesta conterrà i seguenti campi:

- Indirizzo del dispositivo slave a cui è destinato il PDU;
- Indirizzo del dispositivo di classe M0 o M1 che trasmette il PDU;
- Classe di appartenenza del mittente e del destinatario;
- Codice del comando che si vuole eseguire;
- Dati associati alla funzione da eseguire;
- Checksum.

I messaggi di risposta conterranno invece informazioni sull'esito della richiesta. I tipi di risposta previsti sono i seguenti:

- Richiesta espletata completamente;
- Richiesta non eseguita (errore generico);
- Slave busy;
- Richiesta posticipata;
- Errore durante l'elaborazione;
- Funzione non implementata.

Le funzioni previste dal protocollo si possono classificare in diverse categorie:

- **Funzioni di accesso alla memoria:** insieme di tutte le funzioni che permettono la lettura e la scrittura di celle di memoria;
- **Funzioni di lettura e scrittura dei canali di I/O:** insieme di tutte le funzioni che permettono l'accesso in lettura e scrittura dei canali di I/O analogici e digitali;
- **Funzioni di gestione del dispositivo:** insieme di tutte le funzioni che permettono di variare lo stato di funzionamento del dispositivo slave;
- **Funzioni di diagnostica:** insieme di tutte le funzioni che permettono di controllare il corretto funzionamento del dispositivo;
- **Funzioni di protocollo:** insieme di tutte le funzioni che permettono di gestire alcune condizioni particolari di protocollo.

L'interfaccia prevista per il protocollo su linea seriale ha la seguente dichiarazione:

```
int AmicLan_protocollo(unsigned char[] aml_PDU_req,
                       unsigned int num_byte_req,
                       unsigned char[] aml_PDU_resp)
```

dove *aml_PDU_req* è il PDU ricevuto dalla periferica seriale, mentre *aml_PDU_resp* è il PDU di risposta creato dall'application layer del protocollo.

Per il progetto ELIK si è modificata l'implementazione del protocollo in fase di integrazione, in modo che lo stesso si potesse interfacciare sia con il protocollo TCP che con il protocollo USB.

3.4.5 Il protocollo TCP/IP LightWeight IP

Per la parte di sviluppo del protocollo TCP/IP si è scelto di utilizzare un protocollo già esistente per due motivi: il primo è che un protocollo di questo tipo è stato sicuramente sottoposto a diversi test e quindi è garantita la presenza di un minor numero di bug rispetto ad un'implementazione *ad hoc*; il secondo motivo è che, per lo sviluppo di un protocollo complesso quale è il TCP/IP, sarebbero stati necessari diversi mesi uomo non solo per la fase di implementazione, ma anche per la fase di testing.

Dopo un'approfondita indagine sulle possibili opzioni si è scelto di utilizzare il protocollo **lightweight IP** (lwIP), sviluppato da Adam Dunkels dello "Swedish Institute of Computer Science". Questo, fra i protocolli opensource, è quello che offre un miglior rapporto fra completezza e prestazioni; inoltre, essendo pensato per l'utilizzo su microcontrollori, garantisce la gestione dell'elevato flusso di dati proprio dei protocolli internet con il minor dispendio possibile di risorse di memorizzazione.

Nei paragrafi seguenti si descriverà il funzionamento del protocollo, ponendo in maggior rilievo le funzionalità di gestione della memoria, poichè queste costituiscono un requisito fondamentale per l'implementazione di una architettura complessa per microcontrollori.

3.4.5.1 Buffer e memory management

Questo sistema è stato pensato per gestire buffer di dimensioni variabili, dato che il loro contenuto può variare da un segmento TCP di dimensione massima, con centinaia di byte, a piccoli segmenti *ICMP echo reply* di pochi byte. Inoltre, deve essere in grado di far risiedere i dati del buffer in memorie non gestite dal sottosistema di comunicazione, come ad esempio la memoria applicazione o una ROM.

La struttura dati fondamentale per la gestione della memoria è il PBUF, che costituisce la rappresentazione interna di un pacchetto nel lwIP, ed è pensato per stack minimali; esso può essere utilizzato sia per allocare memoria dinamica per la memorizzazione del contenuto di un pacchetto, sia per far risiedere dati di pacchetto nella memoria statica.
Il protocollo lwIP prevede tre tipi di PBUF:

- PBUF RAM: i dati di pacchetto sono memorizzati nella memoria gestita dal sottosistema PBUF;
- PBUF ROM: i dati di pacchetto sono memorizzati in una memoria non gestita dal sottosistema PBUF;
- PBUF POOL: sono PBUF di lunghezza fissa allocati da un pool di PBUF di lunghezza fissa.

Più PBUF possono essere collegati assieme in una lista, chiamata catena di PBUF, e questa può essere costituita da tipi diversi di PBUF.
I tre tipi di PBUF sono stati introdotti per permettere la gestione differenziata della memoria, a seconda delle necessità del protocollo.
I PBUF POOL vengono usati dai driver della periferica di rete, dal momento che le operazioni di allocazione di un singolo PBUF sono veloci e sono dunque preferibili per un utilizzo con interrupt handler.
I PBUF ROM vengono adoperati quando un'applicazione invia dati localizzati nella memoria gestita dall'applicazione. Questi dati potrebbero essere, dal punto di vista dell'applicazione, delle costanti e quindi non essere modificati dopo che il PBUF è stato passato nello stack TCP/IP; quindi il loro principale utilizzo si ha quando i dati sono localizzati in una ROM. Gli headers che vengono appesi ai dati in un PBUF ROM sono memorizzati in un PBUF RAM, che è appeso in testa al PBUF ROM.
I PBUF RAM vengono utilizzati quando un'applicazione invia dati generati dinamicamente. In questo caso, il sistema PBUF alloca

memoria non solo per il dato applicazione, ma anche per l'header che viene appeso al dato. Il sistema PBUF non sa in anticipo quale header sarà appeso al dato e assume quindi il caso peggiore, configurabile a tempo di compilazione.
In sintesi, i PBUF POOL sono PBUF utilizzati per la memorizzazione dei pacchetti in ingresso, mentre i PBUF ROM e i PBUF RAM sono impiegati per i dati in uscita.
La struttura di un PBUF è la seguente:

```
struct pbuf {
  struct pbuf *next;
  void *payload;
  u16_t tot_len;
  u16_t len;
  u16_t flags;
  u16_t ref;
};
```

I campi che caratterizzano la struttura dati in questione sono:

- **Next**: puntatore al successivo PBUF in caso di catene di PBUF;
- **Payload**: puntatore all'inizio dei dati nel PBUF;
- **Len**: lunghezza dei dati contenuti nel PBUF;
- **Tot_len**: somma della lunghezza del PBUF corrente e delle lunghezze di tutti i PBUF successivi nella catena di PBUF;
- **Flags**: tipo di PBUF:
    - *PBUF_FLAG_RAM*: flag che indica che il PBUF è memorizzato nella memoria RAM;
    - *PBUF_FLAG_ROM*: flag che indica che il PBUF è memorizzato nella memoria ROM;
    - *PBUF_FLAG_POOL*: flag che indica che il PBUF è parte di un pool di PBUF;

  - *PBUF_FLAG_REF*: flag che indica che il payload del PBUF riferisce ad una porzione della memoria RAM;
- **Ref**: numero di puntatori che riferiscono a questo PBUF.

La dimensione di un PBUF dipende dall'architettura di riferimento: poiché il microcontrollore utilizzato ha un'architettura a 32 bit con allineamento in 4 byte, la dimensione del PBUF è di 16 byte.

Memory management

Il gestore della memoria che supporta lo schema PBUF è molto semplice: il suo compito è gestire allocazioni e deallocazioni di regioni contigue della memoria e restringere la dimensione di un blocco di memoria allocato precedentemente. Questo gestore utilizza una porzione dedicata della memoria totale nel sistema, il che assicura che il sistema di rete non sfrutti tutta la memoria disponibile compromettendo il funzionamento delle altre applicazioni in esecuzione nel sistema.
Internamente, il gestore della memoria tiene traccia della memoria allocata inserendo una piccola struttura dati in cima a ogni blocco di memoria allocato; inoltre, a ogni blocco è associato un flag che indica il suo stato di allocazione.
Il meccanismo di allocazione della memoria si basa sulla ricerca di blocchi liberi di dimensione pari o maggiore della quantità necessaria. Per la scelta del blocco fra quelli che rispondono alle esigenze richieste viene utilizzato il principio del *first fit*, ossia viene selezionato il primo blocco sufficientemente largo. Quando un blocco di allocazione viene deallocato il flag di allocazione viene resettato. Per evitare la frammentazione della memoria, il gestore esegue un controllo anche sul flag dei blocchi di allocazione precedente e

successivo e, se uno dei due non è utilizzato, i blocchi vengono uniti così da formare un blocco unico più largo.

3.4.5.2 Interfacce di rete

In lwIP i driver di periferica per l'hardware di rete fisico sono rappresentati da una struttura di interfaccia di rete simile a quella del BSD. La struttura dati che la caratterizza è la seguente:

```
struct netif {
  struct netif *next;
  struct ip_addr ip_addr;
  struct ip_addr netmask;
  struct ip_addr gw;
  err_t (*input)(struct pbuf *p, struct netif *inp);
  err_t (*output)(struct netif *netif, struct pbuf *p,
                              struct ip_addr *ipaddr);
  err_t (* linkoutput)(struct netif *netif,
                              struct pbuf *p);
  void *state;
  struct dhcp *dhcp;
  u8_t hwaddr_len;
  u8_t hwaddr[NETIF_MAX_HWADDR_LEN];
  u16_t mtu;
  u8_t flags;
  u8_t link_type;
  char name[2];
  u8_t num;
};
```

Le interfacce di rete vengono mantenute in una lista globale, i cui elementi sono collegati dal puntatore *next* nella struttura.
Ogni interfaccia di rete ha un nome, memorizzato nel campo *name*: questa etichetta di due lettere identifica il tipo di driver di periferica usato per l'interfaccia di rete ed è impiegata solo quando l'interfaccia

è configurata da un operatore a tempo d'esecuzione. Essa viene settata dal driver di periferica e dovrebbe riflettere il tipo di hardware rappresentato dall'interfaccia; poichè il nome non è necessariamente univoco, per differenziare interfacce distinte dello stesso tipo si utilizza il campo *num*.

I tre indirizzi IP *ip_addr*, *netmask* e *gw* sono utilizzati dallo strato IP quando invia e riceve pacchetti (il loro uso è descritto successivamente). Non è possibile configurare un'interfaccia con più di un indirizzo IP e va quindi creata un'interfaccia di rete per ognuno di essi.

Il puntatore *input* indirizza la funzione che verrà richiamata dal driver di periferica alla ricezione del pacchetto.

Un'interfaccia di rete è connessa a un driver di periferica attraverso il puntatore *output*, che indirizza la funzione del driver di periferica, il cui compito è trasmettere un pacchetto nella rete fisica, e viene chiamata dallo strato IP quando deve essere inviato un pacchetto; questo campo viene compilato dalla funzione di inizializzazione del driver di periferica. Il terzo argomento della funzione output, *ip_addr*, è l'indirizzo IP dell'host destinatario del frame del link layer e non è detto che questo corrisponda al campo *destination address* del pacchetto IP. In particolare, quando si invia un pacchetto IP a un host che non si trova nella LAN, il frame viene inviato a un router nella LAN. In questo caso l'indirizzo IP passato alla funzione *output* sarà l'indirizzo IP del router.

Infine, il puntatore *state* punta allo stato specifico del driver di periferica per l'interfaccia di rete ed è settato dal driver.

3.4.5.3 Layer IP

Il lwIP implementa solo le principali funzionalità di IP: esso può inviare, ricevere e inoltrare pacchetti, ma non può inviare o ricevere pacchetti frammentati né gestire pacchetti con il campo option nell'header.

Ricezione di pacchetti

Per i pacchetti IP in ingresso, l'elaborazione inizia quando un driver di periferica richiama la funzione *ip_input*. Innanzitutto viene verificata la versione di IP del pacchetto e la lunghezza dell'header, e inoltre viene calcolato e verificato il checksum dell'header. Ci si aspetta che lo stack non riceva alcun frammento IP, perché si assume che il proxy riassembli ogni pacchetto frammentato; se si verifica ciò, ogni frammento IP ricevuto viene scartato, così come i pacchetti che contengono opzioni nell'header IP (anche in questo caso si assume che vengano gestite dal proxy).

A questo punto, la funzione controlla l'indirizzo di destinazione, confrontandolo con gli indirizzi IP delle interfacce di rete, per determinare se il pacchetto è destinato all'host. Le interfacce di rete sono ordinate in una lista e per la ricerca di elementi al suo interno si adotta un algoritmo lineare, dato che ci si aspetta che il numero di interfacce di rete sia piccolo; per questo motivo, si è scelto di non implementare una strategia di ricerca più complessa.

Se il pacchetto in ingresso è destinato all'host, si analizza il campo *protocol* dell'header per decidere a quale protocollo di livello superiore va trasmesso.

Trasmissione di pacchetti

Un pacchetto in uscita è gestito dalla funzione *ip_output*, che richiama la funzione *ip_route* per trovare l'interfaccia di rete appropriata per la trasmissione del pacchetto nella rete. Quando l'interfaccia è stata determinata, il pacchetto viene passato alla funzione *ip_output_if* che riceve come argomento l'interfaccia di rete individuata. Qui verranno compilati tutti i campi dell'header e verrà calcolato il checksum. Gli indirizzi sorgente e destinazione del pacchetto IP vengono passati come argomento alla funzione *ip_output_if*, anche se l'indirizzo sorgente può essere omesso: in questo caso viene considerato come indirizzo sorgente l'indirizzo IP dell'interfaccia di rete di uscita.

La funzione *ip_route* trova l'interfaccia di rete appropriata ricercandola linearmente nella lista. Durante la ricerca, all'indirizzo di destinazione del pacchetto viene applicata la maschera di rete associata all'interfaccia di rete in analisi: se la rete di destinazione è uguale alla rete dell'interfaccia, essa viene scelta come interfaccia di uscita; altrimenti, se non viene trovata nessuna corrispondenza, viene scelta l'interfaccia di rete di default. Quest'ultima viene configurata all'avvio, ma può essere modificata anche a tempo di esecuzione dall'amministratore di rete. Se l'indirizzo di rete dell'interfaccia di default non corrisponde all'indirizzo della rete di destinazione, il campo *gw* nella struttura *netif* viene scelto come indirizzo IP di destinazione del frame nel link layer.

Dato che i protocolli dello strato di trasporto necessitano dell'indirizzo IP di destinazione per calcolare il checksum di livello di trasporto, l'interfaccia di rete di uscita deve, in alcuni casi, essere determinata prima che il pacchetto venga passato allo strato IP. Ciò viene fatto lasciando allo strato di trasporto il compito di chiamare la funzione *ip_route*; in questo caso, poiché l'interfaccia di rete è già stata determinata nel momento in cui il segmento raggiunge lo strato IP, non è necessario effettuare nuovamente la ricerca. Questi protocolli,

quindi, chiamano direttamente la funzione *ip_output_if*, visto che, ad essa, può essere passata un'interfaccia di rete come parametro.

Inoltro di pacchetti

Se nessuna delle interfacce di rete ha lo stesso indirizzo IP contenuto nel campo *destination address* del pacchetto in ingresso, questo deve essere inoltrato: tale operazione viene eseguita dalla funzione *ip_forward*. Qui, il campo TTL viene decrementato e, se raggiunge zero, viene inviato al mittente del pacchetto un messaggio ICMP di errore e il pacchetto ricevuto viene scartato.
Dopo il decremento del campo TTL è necessario ricalcolare il checksum, poiché l'header IP è cambiato. Fatto ciò, il pacchetto viene inoltrato all'interfaccia di rete appropriata.

ICMP

L'elaborazione del protocollo ICMP è semplice: i pacchetti ICMP ricevuti dalla funzione *ip_input* sono gestiti dalla funzione *icmp_input*, che decodifica l'header ICMP e esegue l'operazione opportuna. Alcuni messaggi ICMP vengono passati ai protocolli di livello superiore che si occuperanno di gestirli con funzioni speciali nello strato di trasporto. Ad esempio, il protocollo dello strato di trasporto UDP può inviare il messaggio ICMP "destination unreachable", e per questo viene usata la funzione *icmp_dest_unreach*.
I messaggi ICMP ECHO sono ampiamente utilizzati per testare una rete e dunque l'elaborazione di tale tipologia di messaggi è ottimizzata per potenziarne le prestazioni. L'elaborazione effettiva avviene nella funzione *icmp_input*, che si occupa di scambiare gli indirizzi IP

sorgente e destinazione del pacchetto in ingresso, di modificare il tipo di ICMP in ECHO REPLY e di aggiornare il checksum.
Il pacchetto viene quindi restituito allo strato IP per la trasmissione.

3.4.5.4 Layer UDP

UDP è un semplice protocollo utilizzato per demultiplare pacchetti tra processi diversi.
Lo stato di ogni sessione UDP è memorizzato in una struttura PCB e ognuna di esse è mantenuta in una lista sulla quale si effettua una ricerca all'arrivo di ogni datagramma.
La struttura di un PCB è la seguente:

```
struct udp_pcb {
  struct ip_addr local_ip;
  struct ip_addr remote_ip;
  u16_t so_options;
  u8_t tos;
  u8_t ttl
  struct udp_pcb *next;
  u8_t flags;
  u16_t local_port, remote_port;
  u16_t chksum_len;
  void (* recv)(void *arg, struct udp_pcb *pcb, struct
                       pbuf *p, struct ip_addr *addr,
                       u16_t port);
  void *recv_arg;
};
```

La struttura PCB dell'UDP contiene un puntatore al PCB successivo nella lista globale dei PCB. Una sessione UDP è definita dagli indirizzi IP e dai numeri di porta degli endpoint, che sono memorizzati nei campi *local_ip*, *remote_ip*, *local_port* e *remote_port*.

Il campo *flag* indica quale politica di checksum UDP viene utilizzata per questa sessione: si può non utilizzare alcuna politica, o utilizzare *UDP Lite* che prevede il calcolo del checksum solo su una parte del datagramma. In quest'ultimo caso, il campo *chksum_len* specifica quanto del datagramma va utilizzato per il calcolo.
Gli ultimi due argomenti *recv* e *recv_arg* si utilizzano quando il datagramma viene ricevuto nella sessione specificata dal PCB. La funzione puntata da *recv* viene richiamata al momento della ricezione di un datagramma.
Vista la semplicità del protocollo UDP, l'elaborazione dell'ingresso e dell'uscita è altrettanto agevole e segue una linea ben precisa.
Per inviare dati, il programma applicazione chiama la funzione *udp_send* che a sua volta richiama la funzione *udp_output*; quest'ultima si occupa di calcolare il checksum e compilare i campi dell'header. Dato che il checksum include l'indirizzo IP sorgente e destinazione del pacchetto IP, in alcuni casi viene chiamata la funzione *ip_route* per trovare l'interfaccia di rete alla quale il pacchetto deve essere trasmesso: l'indirizzo IP di questa viene usato come indirizzo IP sorgente del pacchetto. Infine, il pacchetto è passato alla funzione *ip_output_if* per la trasmissione.
Quando arriva un datagramma UDP, lo strato IP chiama la funzione *udp_input*; qui, se il controllo del checksum è abilitato per questa sessione, viene verificata la sua correttezza e, se la verifica ha avuto esito positivo, il datagramma viene demultiplato. A questo punto, una volta individuato il corrispondente PCB, viene richiamata la funzione puntata da *recv*.

3.4.5.5 Layer TCP

Il TCP è un protocollo dello strato di trasporto che offre un servizio affidabile di trasporto di byte allo strato applicazione. TCP è il protocollo più complesso fra quelli che sono stati già descritti, e il codice che lo implementa costituisce il 50% del codice di lwIP.

Caratteristiche generali

La principale elaborazione del TCP è divisa in sei funzioni:

- *tcp_input*, *tcp_process* e *tcp_receive*: si occupano del flusso in ingresso;
- *tcp_write*, *tcp_enqueue*, e *tcp_output*: riguardano il flusso in uscita.

Quando un'applicazione vuole inviare dati TCP, viene richiamata la funzione *tcp_write*. Questa passa il controllo alla *tcp_enqueue* che si occupa di dividere i dati in segmenti TCP di dimensione appropriata, se necessario, e pone i segmenti in una coda di trasmissione per la connessione. La funzione *tcp_output* controllerà quindi se è possibile inviare dati (ad esempio, se c'è abbastanza spazio nella finestra del ricevente, se la finestra di congestione è larga abbastanza, e così via) e, in caso di esito positivo, invia i dati usando le funzioni *ip_route* e *ip_output_if*.

L'elaborazione di ingresso ha inizio quando la funzione *ip_input*, dopo aver verificato l'header IP, passa un segmento alla funzione *tcp_input*. Essa esegue la verifica di integrità (controllo del checksum e parsing delle opzioni) subito dopo aver deciso a quale connessione TCP appartiene il segmento. Quest'ultimo viene quindi processato dalla funzione *tcp_process*, la quale implementa la macchina a stati finiti del TCP e compie le transizioni di stato necessarie. La funzione *tcp_receive* verrà chiamata se la connessione è in uno stato in cui può

accettare dati dalla rete: in caso affermativo, essa si occuperà di passare il segmento allo strato applicazione. Se il segmento costituisce un ACK per dati unacknowledge (quindi bufferizzati in precedenza), il dato viene rimosso dal buffer e la sua memoria liberata. Inoltre, la ricezione di un ACK implica che il ricevitore è in grado di accettare ulteriori dati e per questo motivo viene chiamata la funzione *tcp_output*.

Strutture dati

Le strutture dati usate nell'implementazione del TCP vengono mantenute piccole per via dei vincoli di memoria dei sistemi minimali per i quali il protocollo lwIP è pensato. Si è cercato di raggiungere un trade-off ottimale fra la complessità delle strutture dati e la complessità del codice che usa le strutture dati, in termini di risparmio di risorse. Per questo motivo la complessità del codice è stata sacrificata a favore di una limitata dimensione dei dati.
Il PCB del TCP è moderatamente largo:

```
struct tcp_pcb {
  struct ip_addr local_ip;
  struct ip_addr remote_ip;
  u16_t so_options;
  u8_t tos;
  u8_t ttl
  struct tcp_pcb *next;
  enum tcp_state state;
  u8_t prio;
  void *callback_arg;
  u16_t local_port;
  u16_t remote_port;
  u8_t flags;
  u32_t rcv_nxt;
  u16_t rcv_wnd;
```

```
  u32_t tmr;
  u8_t polltmr, pollinterval;
  u16_t rtime, mss;
  u32_t rttest;
  u32_t rtseq;
  s16_t sa, sv;
  u16_t rto;
  u8_t nrtx;
  u32_t lastack;
  u8_t dupacks;
  u16_t cwnd;
  u16_t ssthresh;
  u32_t    snd_nxt,
           snd_max,
           snd_wnd,
           snd_wl1,
           snd_wl2,
           snd_lbb;
  u16_t acked;
  u16_t snd_buf;
  u8_t snd_queuelen;
  struct tcp_seg *unsent;
  struct tcp_seg *unacked;
  struct tcp_seg *ooseq;
  err_t (* sent)(void *arg, struct tcp_pcb *pcb,
                            u16_t space);
  err_t (* recv)(void *arg, struct tcp_pcb *pcb,
                            struct pbuf *p, err_t err);
  err_t (* connected)(void *arg, struct tcp_pcb *pcb,
                            err_t err);
  err_t (* accept)(void *arg, struct tcp_pcb *newpcb,
                            err_t err);
  err_t (* poll)(void *arg, struct tcp_pcb *pcb);
  void (* errf)(void *arg, err_t err);
  u32_t keepalive;
  u8_t keep_cnt;
};
```

Dato che le connessioni TCP negli stati LISTEN e TIME-WAIT necessitano di mantenere meno informazioni sullo stato rispetto alle connessioni negli altri stati, per queste si utilizza una struttura dati PCB più piccola:

```
struct tcp_pcb_listen {
  struct ip_addr local_ip;
  struct ip_addr remote_ip;
  u16_t so_options;
  u8_t tos;
  u8_t ttl;
  struct tcp_pcb_listen *next;
  enum tcp_state state;
  u8_t prio;
  void *callback_arg;
  u16_t local_port;
  err_t (* accept)(void *arg, struct tcp_pcb *newpcb,
                                    err_t err);
};
```

I PCB vengono mantenuti in una lista e il puntatore *next* ne collega gli elementi.

La variabile *state* contiene lo stato della connessione TCP corrente, che può essere *Closed*, *Listen*, *Syn_Sent*, *Syn_Rcvd*, *Established*, *Fin_Wait_1*, *Fin_Wait_2*, *Close_Wait*, *Closing*, *Last_Ack*, e *Time_Wait*, come specificato nel flow chart del protocollo.

Nei campi *local_ip, remote_ip*, *local_port* e *remote_port* vengono memorizzati gli indirizzi IP e i numeri di porta che identificano la connessione.

La variabile *mss* contiene il maximum segment size consentito per la connessione.

I campi *rcv_nxt* e *rcv_wnd* si usano quando si riceve un dato: il primo contiene il successivo numero di sequenza atteso dal nodo remoto ed è

pertanto utilizzato quando si inviano ACK all'host remoto; il secondo campo, invece, contiene la finestra del ricevitore annunciata nei segmenti TCP in uscita.
Il campo *tmr* viene usato come timer per le connessioni, che verranno rimosse dopo che la connessione sarà rimasta nello stato TIME-WAIT per un certo periodo di tempo.
Il campo *flags* contiene ulteriori informazioni sullo stato della connessione. Questo campo può assumere i seguenti valori:

- TF_ACK_DELAY: Delayed ACK;
- TF_ACK_NOW: ACK immediate;
- TF_INFR: In fast recovery;
- TF_RESET: La connessione è stata resettata;
- TF_CLOSED: La connessione è stata chiusa con successo;
- TF_GOT_FIN: La connessione è stata chiusa dall'host remoto;
- TF_NODELAY: Disabilita l'algoritmo Nagle.

I campi *rtseq*, *rttest, sa* e *sv* si utilizzano nella stima del RTT: nel primo viene mantenuto il numero di sequenza del segmento che viene utilizzato per stimare il RTT, mentre nel secondo si memorizza l'istante di tempo nel quale è stato inviato il segmento. Il tempo medio di RTT e la sua varianza vengono memorizzati negli ultimi due campi, che vengono usati per calcolare il timeout di ritrasmissione memorizzato nel campo *rto*.
I due campi *lastack* e *dupack* vengono utilizzati nell'implementazione del fast retransmit e fast recovery: il primo contiene il numero di sequenza acknowledged dall'ultimo ACK ricevuto e il secondo contiene il numero di ACK ricevuti per il numero di sequenza contenuto in *lastack*.
La finestra di congestione corrente viene mantenuta nel campo *cwnd* e la threshold dello slow start è memorizzata in *ssthresh*.

I sei campi *snd_ack, snd_nxt, snd_wnd, snd_wl1, snd_wl2* e *snd_lbb* vengono utilizzati quando si inviano dati: nel primo viene memorizzato il più alto numero di sequenza acknowledged dal ricevitore; nel secondo, il numero di sequenza successivo da inviare. La advertised window del ricevitore è mantenuta in *snd_wnd* e i due campi *snd_wl1* e *snd_wl2* si utilizzano quando viene aggiornata *snd_wnd*. L'ultimo campo contiene il numero di sequenza dell'ultimo byte accodato per la trasmissione.
I puntatori a funzione *recv* e *recv_arg* si utilizzano per il passaggio dei dati allo strato applicazione.
Le tre code *unsend*, *unacked* e *ooseq* servono nel momento in cui si inviano e si ricevono dati. I dati che vengono ricevuti dall'applicazione, ma che non sono stati ancora inviati, vengono accodati nella prima, mentre quelli che sono stati inviati, ma che non sono stati riscontrati dall'host remoto, vengono mantenuti in *unacked*; i dati ricevuti fuori sequenza sono invece bufferizzati in *ooseq*.
La struttura *tcp_seq* è la rappresentazione interna di un segmento TCP:

```
struct tcp_seg {
  struct tcp_seg *next;
  struct pbuf *p;
  void *dataptr;
  u16_t len;
  u16_t rtime;
  struct tcp_hdr *tcphdr;
};
```

Questa struttura inizia con un puntatore *next*, usato per collegare i vari segmenti quando vengono accodati.
Il campo *len* contiene la lunghezza del segmento in termini di TCP: ciò significa che questo campo contiene, per un segmento dati, la

lunghezza del dato nel segmento; invece, per un segmento vuoto con il flag SYN o FIN settato, esso varrà uno.
Il campo *p* è il puntatore al buffer contenente il segmento effettivo, mentre i campi puntatore *tcphdr* e *dataptr* indirizzano l'header e il dato nel segmento rispettivamente.
Per i segmenti in uscita, il campo *rtime* è usato per il timeout di ritrasmissione del segmento. Poiché i segmenti in ingresso non necessitano di essere ritrasmessi questo campo non è necessario, per cui, per questo tipo di segmenti, la memoria destinata a questo campo non viene allocata.
La struttura dell'header è la seguente:

```
struct tcp_hdr {
  u16_t src;
  u16_t dest;
  u32_t seqno;
  u32_t ackno;
  u16_t _hdrlen_rsvd_flags;
  u16_t wnd;
  u16_t chksum;
  u16_t urgp;
};
```

Per il significato dei singoli campi dell'header si rimanda all'RFC del protocollo.

Calcolo del numero di sequenza

I numeri di sequenza utilizzati per enumerare i byte nel flusso di informazioni sono interi a 32 bit senza segno, quindi nell'intervallo $[0, 2^{32} -1]$. Dato che il numero di bytes inviati in una connessione TCP può essere maggiore del numero di combinazioni di 32 bit, i numeri di sequenza sono calcolati modulo 2^{32}: ciò significa che gli operatori di

confronto ordinari non si possono utilizzare con i numeri di sequenza TCP. Gli operatori di confronto modificati, chiamati $<_{seq}$ e $>_{seq}$ sono definiti dalla relazione:

$$s <_{seq} t \rightarrow s - t < 0$$
$$s >_{seq} t \rightarrow s - t > 0$$

dove *s* e *t* sono numeri di sequenza.

Accodamento e trasmissione dei dati

Il dato da inviare viene diviso in tronchi di dimensione appropriata e ad ognuno di essi gli viene assegnato un numero di sequenza dalla funzione *tcp_enqueue*. Qui, il dato viene impacchettato in PBUF e racchiuso in una struttura *tcp_seg*. L'header TCP viene costruito nel PBUF e riempito con tutti i campi richiesti, ad eccezione dell'acknowledgement number (*ackno*) e dell'advertised window (*wnd*). Questi campi, infatti, possono cambiare durante il tempo di accodamento del segmento e sono quindi settati dalla funzione *tcp_output* che si occupa dell'effettiva trasmissione del segmento. Dopo che i segmenti vengono costruiti, vengono accodati nella lista *unsent* nel PCB; la funzione *tcp_enqueue* prova a riempire ogni segmento con una quantità di dati pari a MSS e, quando viene trovato un segmento sottodimensionato alla fine della coda *unsent*, questo viene appeso al nuovo dato usando la funzionalità di incatenamento dei PBUF.

Dopo che la funzione *tcp_enqueue* ha formattato e accodato i segmenti, viene richiamata la funzione *tcp_output*, la quale controlla se c'è abbastanza spazio nella finestra corrente per ulteriori dati (la finestra corrente viene calcolata prendendo il massimo della congestion window e della advertised window del ricevitore).

Successivamente, detta funzione riempie i campi dell'header che non sono stati compilati dalla funzione chiamante e trasmette il segmento usando le funzioni *ip_route* e *ip_output_if*. Dopo la trasmissione il segmento viene collocato nella lista *unacked*, nella quale rimane finché non viene ricevuto l'ACK per quel segmento. Quando un segmento si trova in questa lista, gli viene associato un timer per la ritrasmissione.

Quando un segmento viene ritrasmesso, gli header del TCP e dell'IP del segmento originale vengono mantenuti e l'unico vero piccolo cambiamento viene fatto nell'header TCP: i campi *ackno* e *wnd* dell'header TCP vengono aggiornati ai valori correnti dato che potrebbero essere stati ricevuti dei dati fra la trasmissione originale del segmento e la ritrasmissione. Questi cambiamenti (solo due word di 16 bit nell'header) non richiedono di ricalcolare il checksum poiché, per aggiornarlo, si può utilizzare una semplice operazione aritmetica. Inoltre, dal momento che lo strato IP ha già aggiunto l'header IP quando il segmento è stato trasmesso per la prima volta, non vi è alcun motivo di cambiarlo.

Evitare il fenomeno della Silly Window Syndrome

Il SWS è un fenomeno che può comportare un degrado delle prestazioni e che può accadere quando un ricevitore TCP annuncia una finestra piccola e il mittente invia dati per riempire la finestra. Quando questo piccolo segmento viene riscontrato, la finestra si riapre per un piccolo valore e il mittente invia nuovamente un piccolo segmento per riempirla. Ciò genera una situazione in cui il flusso TCP è composto da segmenti molto piccoli.

Per evitare questo fenomeno il ricevente non dovrebbe annunciare aggiornamenti di finestre piccole e il mittente, dal canto suo, non

dovrebbe inviare segmenti di piccole dimensioni quando viene offerta una finestra piccola.
Lato mittente, in lwIP, il SWS si evita senza bisogno di alcun intervento, perché i segmenti TCP vengono costruiti e accodati senza alcuna conoscenza della advertised window del ricevente. In un trasferimento intenso la coda di uscita consiste di un segmento di dimensioni al più pari a MSS. Ciò significa che, se un ricevitore TCP annuncia una piccola finestra, il mittente non manderà il primo segmento nella coda poichè questo è più largo della advertised window, ma aspetterà finché la finestra non sarà larga abbastanza per ricevere un segmento di dimensioni pari a MSS.
In ricezione, il protocollo lwIP evita il problema non annunciando una advertised window più piccola di MSS.

Demultiplexing

Quando i segmenti TCP arrivano alla funzione *tcp_input* vengono demultipati fra i PCB; la chiave di demultiplazione è costituita dall'insieme degli indirizzi IP sorgente e destinazione e dei numeri di porta. Quando si demultipla un segmento vanno distinti due tipi di PCB: quelli che corrispondono a connessioni aperte e quelli che corrispondono a connessioni aperte per metà. Queste ultime sono le connessioni che si trovano nello stato LISTEN e per le quali è specificato solo il numero di porta locale e, opzionalmente, l'indirizzo IP locale; le connessioni aperte sono, invece, quelle in cui sono specificati sia la coppia di indirizzi IP che le porte.
In lwIP, dopo che si è individuato il PCB corrispondente alla richiesta, esso viene spostato in testa alla lista dei PCB. Per connessioni nello stato LISTEN questo spostamento non viene effettuato, dato che ci si aspetta che tali connessioni non ricevano segmenti tanto spesso quanto le connessioni che sono nello stato in cui si ricevono dati.

Ricezione dei dati

L'elaborazione effettiva dei segmenti in ingresso viene fatta dalla funzione *tcp_receive*. Alla ricezione di un pacchetto, l'acknowledgement number del segmento viene confrontato con i segmenti nella coda dei segmenti della connessione non riscontrati: se l'acknowledgement number è maggiore del sequence number di un segmento nella coda *unacked*, quest'ultimo viene rimosso dalla coda e la memoria allocata per esso viene liberata.

Un segmento in ingresso è fuori sequenza se il suo numero di sequenza è maggiore della variabile *rcv_nxt* nel PCB. I segmenti fuori sequenza sono accodati nella coda *ooseq* nel PCB: se il sequence number del segmento in ingresso è uguale a *rcv_nxt*, il segmento viene consegnato allo strato superiore, chiamando la funzione puntata da *recv* nel PCB, e il campo *rcv_nxt* viene incrementato della lunghezza del segmento in ingresso. Poiché la ricezione di un segmento in sequenza potrebbe implicare che uno dei segmenti ricevuti precedentemente fuori sequenza sia il segmento successivo a quello ricevuto, è necessario controllare la coda *ooseq*: se contiene un segmento con sequence number uguale a *rcv_nxt*, esso viene consegnato all'applicazione e il campo *rcv_nxt* viene aggiornato. Questo processo continua finché la coda dei segmenti fuori sequenza non è vuota o il successivo segmento nella stessa è fuori sequenza.

Accettare nuove connessioni

Le connessioni TCP che si trovano nello stato LISTEN sono pronte ad accettare nuove connessioni da un host remoto. Per queste connessioni viene creato un nuovo PCB che va passato al programma applicazione che ha aperto la connessione nello stato LISTEN iniziale. In lwIP ciò si realizza implementando nell'applicazione una funzione di callback, che verrà chiamata quando si stabilisce una nuova connessione.

Quando una connessione nello stato LISTEN riceve un segmento con il flag SYN settato, il sistema crea una nuova connessione e invia un segmento con i flag SYN e ACK settati in risposta al segmento SYN. Quando arriva l'acknowledgement, la connessione entra nello stato ESTABLISHED e viene richiamata la funzione puntata da *accept*.

Fast retransmit

In lwIP sono stati implementati sia il fast retransmit sia il fast recovery, i quali funzionano tenendo traccia dell'ultimo sequence number riscontrato. Se si riceve un altro ACK per lo stesso sequence number il contatore *dupack* nel PCB viene incrementato. Quando questo raggiunge il valore di tre, viene ritrasmesso il primo segmento della coda *unacked* e inizializzato il fast recovery. Quando si riceve un ACK per un nuovo dato il contatore *dupack* viene resettato.

Timer

Il lwIP usa due timer periodici che scattano ogni 200ms e 500ms. Questi due timer vengono usati per implementare timer logici più complessi, come i timer di ritrasmissione, i timer TIME-WAIT e i timer delayed ACK.

Il timer a granularità minore, implementato dalla funzione *tcp_timer_fine*, controlla ogni PCB per vedere se ci sono delayed ACK che devono essere inviati, come indicato nel campo *flag*: se il flag del delayed ACK è settato, viene inviato un segmento di acknowledgment vuoto e il flag viene resettato.

Il timer a granularità maggiore, implementato dalla funzione *tcp_timer_coarse*, controlla anch'esso la lista dei PCB. Per ognuno di essi viene ispezionata la lista dei segmenti non riscontrati e viene incrementata la variabile *rtime*: se essa diventa maggiore dell'attuale

timeout di ritrasmissione memorizzato nella variabile *rto* nel PCB, il segmento viene ritrasmesso e il timeout di ritrasmissione viene raddoppiato. Un segmento viene ritrasmesso solo se consentito dal valore della congestion window e dell'advertised window del ricevente. Dopo la ritrasmissione, la congestion window viene settata a un MSS, la threshold dello slow start viene impostata a metà dell'effettiva dimensione della finestra e lo slow start viene avviato per la connessione.

Per le connessioni che si trovano nello stato TIME-WAIT, questo timer incrementa inoltre il campo *tmr* e, quando quest'ultimo raggiunge la soglia 2xMSL, la connessione viene rimossa. Inoltre, esso incrementa il clock TCP globale: *tcp_ticks*, che viene utilizzato per la stima del RTT e per i timeout di ritrasmissione.

Stima del RTT

La stima del RTT è una parte critica del TCP: il RTT viene infatti utilizzato come parametro per la determinazione di un timeout di ritrasmissione adeguato.

La variabile del PCB *rtseq* contiene il sequence number del segmento per il quale è stato misurato il round trip time. Quando il segmento viene trasmesso per la prima volta, il valore del *tcp_ticks* in quel momento viene memorizzato nella variabile *rttest*. Alla ricezione di un ACK per un numero di sequenza uguale o maggiore di *rtseq*, il RTT viene misurato sottraendo *rttest* dal *tcp_ticks* rilevato alla ricezione del pacchetto nuovo. Se avviene una ritrasmissione durante la misurazione del RTT non viene fatta alcuna misurazione.

Controllo di congestione

L'implementazione del controllo di congestione è sorprendentemente semplice e consiste in poche righe di codice. Alla ricezione di un ACK per nuovi dati, la finestra di congestione, *cwnd*, viene incrementata o di un MSS o di $MSS^2/cwnd$, a seconda che la connessione sia in *slow start* o in *congestion avoidance*. Quando si inviano dati si sceglie il valore minimo fra la advertised window e la congestion window per determinare quanti se ne possano inviare in ogni finestra.

3.4.5.6 AmicLan over TCP/IP

Per sovrapporre il protocollo AmicLan alla pila dei protocolli TCP/IP si è implementata una serie di funzioni che permettono di interfacciare il modulo di livello applicazione con lwIP. La funzione *main*, innanzitutto, richiama la funzione di inizializzazione *amiclan_init*. Questa ha il compito di creare un nuovo socket di tipo LISTEN, al quale assocerà la funzione *amiclan_accept_callback* che si occuperà di eseguire le operazioni opportune nel momento in cui arriverà una richiesta di connessione su questo socket.
Il codice di inizializzazione è il seguente:

```
void amiclan_init(){
  struct tcp_pcb* tcpweb;
  struct tcp_pcb* tcpweb_listen;
  tcpweb = tcp_new();
  if (tcpweb == NULL)
    return;
  if (tcp_bind(tcpweb, IP_ADDR_ANY, 12000) != ERR_OK)
    return;
  tcpweb_listen = tcp_listen(tcpweb);
  if (tcpweb_listen == NULL){
    tcp_abort(tcpweb);
```

```
    tcpweb = NULL;
    return;
  }
  tcpweb = tcpweb_listen;
  tcp_accept(tcpweb, amiclan_accept_callback);
}
```

La funzione di inizializzazione, per prima cosa, richiama la funzione *tcp_new* per creare un PCB senza inserirlo in alcuna coda; successivamente, utilizzando la funzione *tcp_bind*, lo imposta per ricevere connessioni da qualsiasi indirizzo IP alla porta 12000. Questo PCB viene poi passato come argomento alla funzione *tcp_listen*, che permette di crearne uno nuovo di tipo LISTEN al quale, tramite la funzione *tcp_accept*, verrà associata la funzione di callback appropriata.

Appena il sistema riceve una richiesta di connessione da parte di un host remoto, viene creato un nuovo PCB e richiamata la funzione di callback che avrà il compito di inizializzare la nuova connessione, impostando le variabili utilizzate come parametri di comunicazione e associandole le funzioni di callback appropriate. Per la comunicazione dei dati in transito fra Transport Layer e Application Layer si utilizza una struttura dati che ha la seguente forma:

```
struct DataConnection {
  char* DataRecv;
  u16_t DataRecvLen;
  char* DataToSendPtr;
  u16_t LeftToSend;
  u16_t TotalToSend;
  u16_t SentAmountToAcknowledge;
  u16_t failed;
};
```

I campi *DataRecv* e *DataRecvLen* contengono, rispettivamente, i dati ricevuti dallo strato di trasporto e la loro dimensione.
I campi *DataToSendPtr*, *LeftToSend*, *TotalToSend* e *SendAmountToAcknowledge* contengono invece i dati da trasmettere e alcune informazioni utili alla trasmissione.
L'implementazione della funzione *amiclan_accept_callback* è riportata di seguito:

```
err_t amiclan_accept_callback(void *arg, struct tcp_pcb
                                    *pcb, err_t err)
{
  struct DataConnection *ws;
  tcp_setprio(pcb, TCP_PRIO_MAX);
  tcp_mss(pcb) = MSS;
  ws = mem_malloc(sizeof(struct DataConnection));
  if(ws == NULL)
    return ERR_MEM;
  tcp_arg(pcb, ws);
  tcp_recv(pcb, amiclan_recv_callback);
  tcp_sent(pcb, amiclan_sent_callback);
  tcp_poll(pcb, amiclan_poll_callback, 1);
  return ERR_OK;
}
```

Questa funzione, dopo le opportune impostazioni del PCB, alloca lo spazio di memoria per la variabile di comunicazione e la imposta come parametro delle funzioni di callback mediante la funzione *tcp_arg*, poi imposta le funzioni da richiamare in caso di ricezione e di consegna del pacchetto e la funzione di polling, che viene richiamata, in questo caso, due volte al secondo.
La funzione di polling *amiclan_poll_callback* si occupa semplicemente di controllare nella struttura *DataConnection* se ci sono ancora dei dati da trasmettere e, in caso affermativo, richiama la funzione *send_buf* che si occupa di trasmetterli.

La funzione *amiclan_sent_callback* aggiorna il campo *SentAmountToAcknowledge* rimuovendo il numero di byte riscontrati e controlla se ci sono altri dati da inviare: in caso affermativo richiama la funzione *send_buf*, altrimenti reinizializza la struttura dati *DataConnection*.

La funzione *amiclan_recv_callback* inizializza la struttura dati *DataConnection* con i dati ricevuti dal layer TCP, invia il riscontro richiamando la funzione *tcp_recved*, libera la memoria dei PBUF mediante la funzione *pcb_free* e richiama il protocollo AmicLan, trasmettendogli come parametro la struttura dati inizializzata.

Per comprendere meglio la struttura delle precedenti funzioni si riporta di seguito la funzione *send_buf*.

```
void send_buf(struct tcp_pcb *pcb, struct
                        WebConnection *ws) {
  err_t errorvalue;
  int amounttosend = MSS;
  if (ws->LeftToSend < amounttosend)
    amounttosend = ws->LeftToSend;
  if (amounttosend == 0)
    return;
  errorvalue = tcp_write(pcb, ws->DataToSendPtr,
                              amounttosend, 1);
  <Controlla errori>
  ws->DataToSendPtr += amounttosend;
  ws->LeftToSend -= amounttosend;
}
```

Dopo aver effettuato i necessari controlli sulle variabili della struttura di comunicazione, la funzione descritta richiama la funzione *tcp_write* che accoda il dato per l'invio. Infine, aggiorna le variabili di trasmissione.

Il codice del protocollo AmicLan non ha richiesto modifiche sostanziali, se non per quanto riguarda la funzione di trasmissione dei dati, che in questo caso si occuperà di inizializzare la struttura dati di tipo *DataConnection*, utilizzata per la comunicazione fra i due protocolli. Inoltre, i parametri passati non sono dati di tipo *char*, ma puntatori al PCB della connessione e alla struttura dati di comunicazione. L'interfaccia prevista per il protocollo su TCP/IP ha la seguente dichiarazione:

```
int AmicLan_protocollo(struct tcp_pcb * pcb,
                       struct DataConnection * ws)
```

3.4.6 Il protocollo USB lato ARM 7

Prima di descrivere l'implementazione del driver USB per la scheda LPC2378, si ritiene indispensabile fornire una breve descrizione del funzionamento della periferica seriale in questione, rimandando alla documentazione specifica per quanto riguarda lo strato fisico. Seguirà poi la descrizione delle scelte implementative per il microcontrollore sul quale si è lavorato.

3.4.6.1 Descrizione generale

La rete USB supporta tre velocità di comunicazione: *Low speed* alla bitrate di 1.5 Mbps, usata per periferiche semplici; *Full speed* alla bitrate di 12 Mbps, per la maggior parte delle periferiche; *High speed* alla bitrate di 480 Mbps, utilizzata per lo stream audio e video.
Sebbene una rete USB, dal punto di vista fisico, coinvolga degli hub e quindi abbia una struttura ad albero, dal punto di vista logico essi non introducono alcuna complessità e, per questo, la rete può essere vista come strutturata a stella.

Essendo il protocollo USB di tipo Plug&Play, il master deve poter conoscere, all'atto del collegamento, la bitrate richiesta per comunicare con la nuova periferica: ciò si ottiene inserendo una resistenza di pull-up o alla linea D+, comunicando così al master la presenza di una periferica *Full Speed*, o alla linea D-, e in questo caso il master riconosce una periferica *Low speed*.

Una volta riconosciuta la velocità di comunicazione, il master può iniziare a trasferire dati da e verso la nuova periferica. Questi dati vengono trasmessi su una serie di connessioni logiche dette *pipe*. Una pipe ha origine da un buffer nel master ed è connessa ad una periferica remota con uno specifico indirizzo; questa termina dentro la periferica in un'entità astratta detta *endpoint*, costituita da un buffer dove vengono memorizzati i dati e da un interrupt che segnala alla CPU l'arrivo di un nuovo pacchetto.

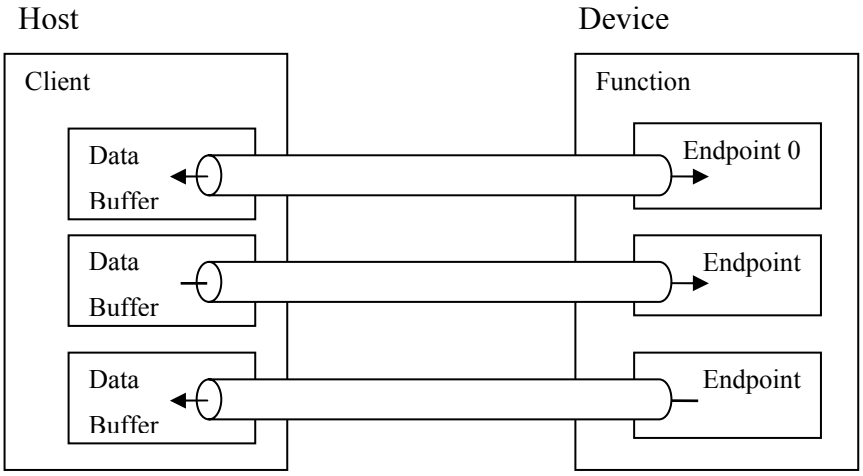

Figura 33: Esempio di struttura logica di una comunicazione USB

Queste pipe sono implementate, nel bus seriale, a divisione di tempo. Ogni millisecondo il master invia un token (Start Of Frame) per dividere i 12 Mbps (nel caso di periferica *Full Speed*) in una serie di frame. A ogni pipe, per la trasmissione dei dati, è allocato uno slot

temporale in ogni frame, la cui lunghezza varia a seconda della tipologia di trasmissione.

Il protocollo USB supporta quattro tipi diversi di pipe con differenti caratteristiche di trasferimento: *control*, *interrupt*, *bulk* e *isochronous*. Tutte le pipe sono monodirezionali, ad eccezione della pipe di controllo che è bidirezionale ed è dedicata alla trasmissione di informazioni di configurazione, su richiesta del master, ed è associata all'*endpoint zero*; quindi, nello sviluppo di un'applicazione slave, questo *endpoint* deve essere sempre presente.

La pipe di tipo *interrupt* permette la comunicazione asincrona fra master e slave: dal momento che nessuna periferica di rete può iniziare un trasferimento di dati senza esplicita richiesta da parte del master, essa permette di definire richieste periodiche (con periodo variabile da 1 a 255 msec) di trasferimento dati da parte del master; in buona sostanza, l'*interrupt pipe* definisce un polling rate.

La pipe di tipo *isochronous* viene utilizzata per il trasferimento di dati in tempo reale: essa non effettua controllo di errore e invia i dati a prescindere dal fatto che il pacchetto precedente sia stato consegnato con successo o meno. Un'importante caratteristica di questo tipo di pipe è che trasferisce i dati ad un rate costante.

La pipe di tipo *bulk* si utilizza per il trasferimento di dati che non rientrano nelle categorie descritte precedentemente. Questi vengono trasmessi allo stesso modo e con le stesse dimensioni della *interrupt pipe*, con la differenza che la *bulk pipe* non ha un polling rate definito, ma utilizza la banda lasciata disponibile dalle altre pipe eventualmente presenti. Se il bus è *busy*, quindi, un trasferimento bulk può essere ritardato; al contrario, se il bus è *idle*, in un singolo frame possono avvenire diversi trasferimenti di questo tipo, a differenza delle pipe *interrupt* e *isochronous* nelle quali i trasferimenti sono invece limitati a un massimo di un pacchetto per frame.

Per quanto riguarda il protocollo di trasferimento dei dati, il bus si delinea in frame inviando uno *Start Of Frame* ogni millisecondo e le pipe trasferiscono i pacchetti di dati all'interno di questi frame. Ogni trasferimento di dati è costituito da tre transazioni di pacchetto: *Token Phase*, in cui viene definito quale tipo di transazione sta avendo luogo, *Data Packet Phase*, in cui vengono trasferiti i dati di interesse e *Handshake Packet Phase*, in cui si stabilisce se il trasferimento è avvenuto con successo.

Quando viene connessa una periferica al dispositivo master, subito dopo la fase di individuazione della sua velocità di segnalazione le viene assegnato l'indirizzo zero: in questo modo il master avrà un canale di controllo disponibile all'indirizzo zero *endpoint zero*. La pipe di controllo viene poi utilizzata dal master per ricavare i requisiti della nuova periferica e aggiungerla alla rete: questo processo viene detto *Enumerazione*. I dati richiesti dal master durante questa fase vengono mantenuti in una gerarchia di *descrittori*, che sono degli array di dati da trasferire al master in risposta alle richieste di enumerazione. Il numero minimo di descrittori richiesti comprende: un *device descriptor*, un *configuration descriptor*, un *interface descriptor* e tre *endpoint descriptor* (uno di controllo, una pipe di ingresso e una di uscita).

Il *device descriptor* contiene le informazioni base sulla periferica, fra le quali il Vendor ID, il Product ID e la dimensione massima del pacchetto (Tabella 3).

Tabella 3: Formato del device descriptor

Offset (Dec)	Field	Size	Description
0	bLength	1	Dimensione del descrittore in byte
1	bDescriptorType	1	Tipo di descrittore DEVICE (01h)
2	bcdUSB	2	Numero di release delle specifiche USB (BCD)
4	bDeviceClass	1	Codice di classe

5	bDeviceSubclass	1	Codice di sottoclasse
6	bDeviceProtocol	1	Codice di protocollo
7	bMaxPacketSize(0)	1	Dimensione massima del pacchetto per l'Endpoint zero
8	idVendor	1	Vendor ID
10	idProduct	1	Product ID
12	bcdDevice	1	Numero di release della periferica (BCD)
14	iManufacturer	1	Indice del descrittore di stringa per il produttore
15	iProduct	1	Indice del descrittore di stringa per il prodotto
16	iSerialNumber	1	Indice del descrittore di stringa contenente il numero seriale
17	bNumConfigurations	1	Numero delle possibili configurazioni

Il *configuration descriptor* contiene informazioni sui requisiti di potenza e il numero di interfacce supportate (Tabella 4).

Tabella 4: Formato del configuration descriptor

Offset (Dec)	Field	Size	Description
0	bLength	1	Dimensione del descrittore in byte
1	bDescriptorType	1	Tipo di descrittore CONFIGURATION (02h)
2	bTotalLength	2	Dimensione di tutti i dati restituiti per questa configurazione in byte; include la combinazione di tutte le lunghezze di tutti i descrittori per questa configurazione
4	bNumInterface	1	Numero di interfacce supportate dalla configurazione
5	bConfigurationValue	1	Identificatore per richieste Set_Configuration e Get_Configuration
6	iConfiguration	1	Indice del descrittore di stringa per la configurazione
7	bmAttributes	1	Impostazioni di self/bus power e remote wakeup
8	MaxPower	1	Requisiti di potenza del bus, espresse in [milliAmpere/2]

L'*interface descriptor* descrive una collezione di *endpoint* e contiene informazioni sul gruppo di pipe associate ad un particolare task (Tabella 5).

Tabella 5: Formato dell'interface descriptor

Offset (Dec)	Field	Size	Description
0	bLength	1	Dimensione del descrittore in byte
1	bDescriptorType	1	Tipo di descrittore INTERFACE (04h)
2	bInterfaceNumber	1	Numero che identifica l'interfaccia
3	bAlternateSettings	1	Valore utilizzato per selezionare un'impostazione alternativa
4	bNumEndPoints	1	Numero di Endpoint supportati, escluso l'Endpoint 0
5	bInterfaceClass	1	Codice di classe
6	bInterfaceSubclass	1	Codice di sottoclasse
7	bInterfaceProtocol	1	Codice di protocollo
8	iInterface	1	Indice del descrittore di stringa per l'interfaccia

L'ultimo descrittore trasferisce i dettagli di configurazione di ogni *endpoint* supportato da una determinata interfaccia, come il tipo di trasferimento supportato, la dimensione massima del pacchetto, il numero dell'*endpoint* e il polling rate nel caso di *interrupt pipe* (Tabella 6).

Tabella 6: Formato dell'endpoint descriptor

Offset (Dec)	Field	Size	Description
0	bLength	1	Dimensione del descrittore in byte
1	bDescriptorType	1	Tipo di descrittore ENDPOINT (05h)
2	bEndpointAddress	1	Numero dell'Endpoint e direzione
3	bmAttributes	1	Tipo di trasferimento supportato
4	wMaxPacketSize	2	Dimensione massima di pacchetto supportata
5	bInterval	1	Latenza massima/intervallo di polling/NAK rate

La fase di enumerazione inizia con il trasferimento da parte del master di un pacchetto SETUP, seguito da un pacchetto dati che contiene i codici di comando per le azioni di controllo che il master vuole inviare. Conclusa l'Handshake Packet Phase il master invia un pacchetto IN e la periferica USB restituirà un pacchetto dati

contenente le informazioni richieste (nella fattispecie, un descrittore). Quindi il processo di enumerazione esegue almeno le seguenti richieste:

- Il master preleva, utilizzando la dimensione di pacchetto più piccola possibile, il vendor ID, il product ID e la dimensione massima di pacchetto. I trasferimenti successivi utilizzeranno questa dimensione come dimensione del pacchetto;
- Una volta ottimizzata la connessione con l'Endpoint zero il master invia un comando RESET al nuovo nodo per assicurarsi che questo si trovi in uno stato conosciuto;
- Dato che tutte le nuove periferiche vengono accedute dal master all'indirizzo zero, questo va mantenuto libero; il passo successivo è quindi assegnare un nuovo indirizzo univoco alla periferica;
- Per concludere il processo di enumerazione, il master richiede tutti i descrittori.

Conclusa la fase di enumerazione l'interfaccia è configurata e pronta all'uso.

Le classi

Le classi che costituiscono le definizioni delle periferiche di comunicazione sono tre: la *Communication Device Class* è una definizione a livello di periferica usata dall'host per identificare correttamente una periferica di comunicazione, che può presentare differenti tipi di interfaccia; la *Communication Interface Class* definisce un meccanismo general-purpose che può essere utilizzato per abilitare tutti i tipi di servizi di comunicazione dell'USB; la *Data Interface Class* definisce un meccanismo general-purpose per abilitare

trasferimenti di tipo bulk o isochronous nell'USB, qualora i dati non rispettino i requisiti per nessuna delle classi esistenti.

Il compito dell'interfaccia *Communication Class* è gestire le richieste e le notifiche che controllano e configurano gli stati operazionali della periferica, nonché notificare all'host il verificarsi di eventi nella periferica stessa. La trasmissione dei dati è ottenuta usando le interfacce *Data Class* in collaborazione con l'interfaccia *Communication Class*. Queste interfacce possono usare qualsiasi classe USB già definita o possono essere vendor-specific.

La *Communication Class* è quindi un'interfaccia di gestione, necessaria per tutte le periferiche di comunicazione, mentre l'interfaccia *Data Class* può essere usata per trasportare dati i cui usi e struttura non siano definiti da alcuna classe. Il formato dei dati in transito su quest'interfaccia può essere identificato usando l'interfaccia *Communication Class* associata.

La *Communication Class* definisce un'interfaccia che consiste in un elemento di gestione e eventualmente in uno di notifica. L'elemento di gestione configura e controlla la periferica e consiste nell'*endpoint zero*. L'elemento di notifica trasporta eventi sotto forma di messaggi in formato standard verso l'host e consiste in un *endpoint* di tipo *interrupt* o *bulk*.

La *Data Class* definisce un'interfaccia dati come interfaccia standard con tipo di classe *Data Class*. Come già detto, essa permette di trasmettere dati in un formato non definito: è quindi responsabilità del software host e della periferica scambiarsi informazioni su altre interfacce (come l'interfaccia *Communication Class*) per determinare il formato da usare. Questo tipo di interfaccia può gestire solo *endpoint* di tipo *isochronous* o *bulk*.

Tutte le periferiche di comunicazione saranno dunque caratterizzate da un'interfaccia *Communication Class* più zero o più interfacce di trasmissione dati.

Nelle specifiche di protocollo sono riportati diversi modelli di periferica, dove con il termine "modello" si intende la descrizione del tipo di periferica e delle interfacce che ne permettono la realizzazione.
Per lo sviluppo di particolari *Communication Class* è stato introdotto un quarto tipo di descrittore, il *functional descriptor*, che descrive il contenuto delle informazioni class-specific all'interno di un *interface descriptor*. Tali descrittori iniziano tutti con un *header descriptor*, che consente al software dell'host di individuare facilmente i descrittori class-specific. Ognuno di essi consiste in uno o più *functional descriptor*. L'*header descriptor* ha la seguente forma:

Tabella 7: Formato dell'header descriptor

Offset (Dec)	Field	Size	Description
0	bFunctionLength	1	Dimensione del descrittore in byte
1	bDescriptorType	1	Tipo di descrittore CS_INTERFACE (24h)
2	bDescriptorSubtype	1	Sottotipo dell'*header functional descriptor*
3	bcdCDC	2	Numero di release delle specifiche USB Class Definition for Communication Device

Per la struttura dei diversi *functional descriptor* si rimanda alle specifiche di classe.

3.4.6.2 Implementazione

Per poter implementare un sistema di comunicazione su USB è necessario, prima di tutto, scegliere le classi che andranno a costituire la definizione delle periferiche di comunicazione. Come visto nel paragrafo 3.4.6.1, le specifiche offrono diversi modelli di periferica.
Fra le possibili scelte implementative offerte dalla specifica USB, si è deciso di sviluppare il firmware come porta COM virtuale, secondo il modello "Abstract Control Model Serial Emulation". Si è deciso ciò

per diverse ragioni, non tanto di carattere implementativo, quanto di tipo economico.

La prima motivazione è che l'obiettivo di ELIK è quello di immettere sul mercato un prodotto a basso costo: per far ciò è necessario abbassare il più possibile i *non recurrent engineering costs*. La scelta di una soluzione diversa dalla COM virtuale avrebbe aumentato notevolmente i costi NRE, dato che avrebbe richiesto l'acquisto di ambienti di sviluppo per l'implementazione dei driver del master e quindi una maggior spesa da ammortizzare, nonché lo studio degli stessi e quindi l'aumento dei mesi uomo richiesti per lo sviluppo del sistema. La seconda ragione è che, almeno fino allo sviluppo dei driver *ad hoc* per il master, non si sarebbe potuto effettuare il testing del protocollo sulla scheda; invece, in questo modo, si è potuto utilizzare il software di testing del protocollo AmicLan *Heron* (vd. Paragrafo 3.4.7), già presente in azienda. Un ulteriore fattore che ha portato ad una scelta di questo tipo è l'esistenza di diversi esempi di implementazione che utilizzano questo approccio, fra i quali il sistema *lpcusb* di Bertrik Sikken, da cui si è preso maggiormente spunto.

Per gestire le operazioni sulla scheda sarà possibile utilizzare il driver *usbser.sys*, offerto dalla Microsoft per i sistemi operativi Windows 95/98/ME/2000/XP, che consente di visualizzare la periferica come porta COM standard. Per il sistema operativo Linux, invece, la porta seriale USB è gestita dal modulo di kernel *cdc_acm*, che permette di visualizzare la scheda come */dev/ttyACMx*.

Il modello scelto è costituito da un'interfaccia *Communication Class* e da un'interfaccia *Data Class*.

L'interfaccia *Communication Class* consiste di due *pipe*: una viene usata per implementare l'elemento di gestione e l'altra per implementare l'elemento di notifica.

L'interfaccia *Data Class* usa due *pipe* per implementare i canali sui quali verranno trasportati i dati in un formato non specificato.

Le pipe di gestione e notifica sono implementate mediante i *functional descriptor* specifici per la sottoclasse *Abstract Control Model*. La pipe di gestione è descritta mediante il descrittore *Call Management Functional Descriptor*, mentre la pipe di notifica mediante il descrittore *ACM Functional Descriptor*.
Il *device descriptor* contiene le seguenti informazioni:

Tabella 8: Valori dei campi del device descriptor

Field	Contenuto
bLength	0x12
bDescriptorType	0x1
bcdUSB	0x0200
bDeviceClass	0x02
bDeviceSubclass	0x00
bDeviceProtocol	0x00
bMaxPacketSize(0)	64
idVendor	*xxxx*
idProduct	*xxxx*
bcdDevice	0x0100
iManufacturer	0x01
iProduct	0x02
iSerialNumber	0x03
bNumConfigurations	0x01

Questo descrittore contiene informazioni di carattere generale. Il campo *bcdUSB* informa che il sistema si basa sulle specifiche USB 2.0, mentre i campi *bDeviceClass*, *bDeviceSubclass* e *bDeviceProtocol* per questo tipo di descrittore sono standard. Per la descrizione degli altri campi si rimanda al Paragrafo 3.4.6.1.
Il *configuration descriptor* contiene le seguenti informazioni:

Tabella 9: Valori dei campi del configuration descriptor

Field	Contenuto
bLength	0x09
bDescriptorType	0x02
bTotalLength	67
bNumInterface	0x02
bConfigurationValue	0x01
iConfiguration	0x00
bmAttributes	0xC0
MaxPower	0x32

Questo descrittore è previsto allo scopo di informare l'host sul fatto che il sistema è caratterizzato da due interfacce, che è *self-powered* e che il consumo massimo di potenza è pari a 100 mA.

La prima interfaccia ad essere descritta è quella per il controllo della classe.

L'*interface descriptor* relativo contiene le seguenti informazioni:

Tabella 10: Valori dei campi dell'interface descriptor

Field	Contenuto
bLength	0x09
bDescriptorType	0x04
bInterfaceNumber	0x00
bAlternateSettings	0x00
bNumEndPoints	0x01
bInterfaceClass	0x02
bInterfaceSubclass	0x02
bInterfaceProtocol	0x01
iInterface	0x00

In questo descrittore si informa che, a questa interfaccia, è associato un solo endpoint; inoltre esso notifica che l'interfaccia appartiene alla sottoclasse *Abstract Control Model* e che segue il protocollo standard *V.25ter*, costituito dall'insieme dei comandi AT comuni.

L'*header functional descriptor* contiene le seguenti informazioni:

Tabella 11: Valori dei campi del'header functional descriptor

Field	Contenuto
bFunctionLength	0x05
bDescriptorType	0x24
bDescriptorSubtype	0x00
bcdCDC	0x0110

Il compito di questo descrittore è introdurre i *functional descriptor*, informando sulla versione delle specifiche CDC seguite.

Il *Call Management Functional Descriptor* contiene le seguenti informazioni:

Tabella 12: Valori dei campi del Call Management Functional Descriptor

Field	Contenuto
bFunctionLength	0x05
bDescriptorType	0x24
bDescriptorSubtype	0x01
bmCapabilities	0x01
bDataInterface	0x01

Il byte *bmCapabilities* indica che la periferica gestisce il *Call Management* essa stessa, mentre il byte *bDataInterface* imposta l'interfaccia della *Data Class* come interfaccia alternativa per la gestione del *Call Management*.

L' *ACM Functional Descriptor* contiene le seguenti informazioni:

Tabella 13: Valore dei campi dell'ACM Functional Descriptor

Field	Contenuto
bFunctionLength	0x04
bDescriptorType	0x24
bDescriptorSubtype	0x02
bmCapabilities	0x02

Questo descrittore ha lo scopo di informare sui comandi supportati dall'interfaccia *Communication Class*.
I comandi supportati sono *Set_Line_Coding*, *Set_Control_Line_State*, *Get_Line_Coding* e la notifica del *Serial_State*.
L'*Union Functional Descriptor* contiene le seguenti informazioni:

Tabella 14: Valore dei campi dell'Union Functional Descriptor

Field	Contenuto
bFunctionLength	0x05
bDescriptorType	0x24
bDescriptorSubtype	0x06
bMasterInterface	0x00
bSlaveInterface0	0x01

Questo tipo di descrittore descrive la relazione esistente fra gruppi di interfacce che possono essere considerate costituenti un'unità funzionale; il suo compito è quindi quello di designare un'interfaccia master all'interno dell'unità funzionale e le relative interfacce slave.
L'*Endpoint Notification Descriptor* contiene le seguenti informazioni:

Tabella 15: Valore dei campi dell'Endpoint Notification Descriptor

Field	Contenuto
bLength	0x07
bDescriptorType	0x05
bEndpointAddress	0x81
bmAttributes	0x03
wMaxPacketSize	8
bInterval	0x0A

Il campo *bEndPointAddress* indica che l'endpoint descritto è di uscita e ha l'indirizzo 1. Il campo *bmAttribute* indica che il tipo di trasferimento è *interrupt*, mentre il campo *bInterval* specifica un intervallo di polling di 10 frame (10 msec). La dimensione massima di pacchetto trasmesso è 8 byte.

L'*interface descriptor* per la *Data Class* contiene le seguenti informazioni:

Tabella 16: Valore dei campi dell'Interface Descriptor per la Data Class

Field	Contenuto
bLength	0x09
bDescriptorType	0x04
bInterfaceNumber	0x01
bAlternateSettings	0x00
bNumEndPoints	0x02
bInterfaceClass	0x0A
bInterfaceSubclass	0x00
bInterfaceProtocol	0x00
iInterface	0x00

A quest'interfaccia di tipo *Data Interface Class* (come specificato nel campo *bInterfaceClass*) sono associati i due endpoint che seguono.
Gli *endpoint descriptor* per la *Data Class* contengono le seguenti informazioni:

Tabella 17: Valore dei campi degli Endpoint Descriptor

Field	Contenuto
bLength	0x07
bDescriptorType	0x05
bEndpointAddress	0x05
bmAttributes	0x02
wMaxPacketSize	64
bInterval	0x00

Field	Contenuto
bLength	0x07
bDescriptorType	0x05
bEndpointAddress	0x82
bmAttributes	0x02
wMaxPacketSize	64
bInterval	0x00

Entrambi gli endpoint hanno una dimensione massima di pacchetto di 64 byte e la loro modalità di trasferimento è di tipo *bulk* (per questo motivo il campo *bInterval* non è specificato). L'unica cosa che li differenzia è che il primo è un endpoint di uscita e ha indirizzo 0x5, mentre il secondo è un endpoint di ingresso con indirizzo 0x2.

L'esecuzione del codice sorgente è incentrata sul gestore delle interruzioni relative alla periferica USB. Per poter illustrare in dettaglio il funzionamento del firmware sarebbe necessario conoscere a fondo il protocollo di messaggistica USB, per cui ci si limiterà a descrivere il funzionamento generale, senza entrare nello specifico dell'implementazione.
Il codice si può suddividere in tre parti fondamentali: una parte di inizializzazione comprendente, oltre alla dichiarazione dei vari *descrittori*, anche le funzioni che permettono di inizializzare le strutture dati utilizzate durante l'esecuzione; una seconda parte di comunicazione con l'interfaccia *Serial Interface Engine* (il blocco hardware che implementa il layer di protocollo USB) e di "smistamento" dei messaggi ricevuti, a seconda del tipo di messaggio; infine, un'ultima parte dove vengono implementati i protocolli di comunicazione ad alto livello.
L'esecuzione inizia con l'installazione dei vari endpoint, intesa come l'allocazione della memoria utilizzata come buffer di comunicazione e delle funzioni handler che si occuperanno di gestire le richieste. E' stato implementato un handler per ogni endpoint. L'handler che gestisce l'endpoint di notifica differisce dagli altri poiché non è richiamato direttamente da un evento sull'endpoint 1, essendo questo un endpoint associato ad una pipe di uscita.
Al termine di questa fase viene richiamata in polling la funzione di gestione degli interrupt, che può essere suddivisa in tre parti: una prima parte che si occupa degli eventi relativi allo stato della

periferica (che non verrà trattata nel dettaglio), una parte che controlla e gestisce gli interrupt sugli endpoint e una parte che gestisce l'interrupt generato dall'arrivo di uno SOF (il quale, anche se gestito nell'implementazione, non provoca l'esecuzione di operazioni significative).

La parte di gestione degli interrupt sugli endpoint controlla, al verificarsi di un'interruzione, quali dei 32 endpoint gestibili dall'LPC2378 l'hanno generata e il tipo di interruzione e, successivamente, richiama l'handler dell'endpoint.

Per quanto riguarda gli endpoint per la *Data Class* vengono richiamate le funzioni *BulkIn* o *BulkOut* a seconda che l'endpoint sia d'ingresso o di uscita.

Invece, per l'*endpoint zero*, viene richiamata la funzione *USBHandleControlTransfer* che si occupa di gestire non solo le richieste di controllo, ma anche i trasferimenti relativi all'*Abstract Control Model*. Nel caso di richiesta standard richiama la funzione *USBHandleStandardRequest*, che a sua volta richiama la funzione specifica per il tipo di richiesta standard pervenuta, che si occupa di generare il messaggio di risposta; nel caso di richiesta di tipo classe, per ottenere il messaggio di risposta, viene richiamata la funzione *HandleClassRequest* che ha il compito di generare il messaggio di risposta appropriato. Non appena ottenuto il messaggio di risposta dalla funzione specifica, la funzione *USBHandleControlTransfer* la invia sull'interfaccia SIE e il sistema è pronto per gestire una nuova richiesta.

3.4.6.3 AmicLan over USB

L'implementazione di AmicLan over USB ha previsto lo sviluppo delle funzioni *handler* per gli endpoint della *Data Class* in modo che si potessero interfacciare con il codice del protocollo AmicLan.
Il prototipo della funzione *AmicLan_protocollo* è lo stesso del protocollo AmicLan su linea seriale.
La funzione *BulkOut* sarà scritta nel seguente modo:

```
static void BulkOut(U8 bEP, U8 bEPStatus)
{
  < Sezione Dichiarazioni >
  receive_counter += USBHwEPRead(bEP, &aml_PDU_req,
                       MAXDIM_RX – receive_counter_0);
  AmicLan_protocollo(aml_PDU_req, receive_counter,
                       aml_PDU_resp);
}
```

Mentre la funzione *BulkIn* eseguirà le seguenti operazioni:

```
static void BulkIn(U8 bEP, U8 bEPStatus)
{
  if (tx_counter_0 > 0)
    USBHwEPWrite(bEP, tx_buffer, tx_counter);
  tx_counter = 0;
}
```

dove *tx_counter* e *tx_buffer* sono variabili globali impostate dalla funzione *AmicLan_protocollo* e contenenti il PDU di risposta e la sua dimensione in byte.

3.4.7 Testing del firmware

Per il testing del firmware si è scelto di utilizzare software freeware o shareware, cosicché questa fase non gravasse sui costi NRE.
Per il testing del protocollo ModBus sono stati utilizzati i seguenti applicativi:

- **Modpoll Modbus® Polling Tool**: simulatore da riga di comando, freeware, di un master ModBus, basato sulle librerie FieldTalk™ Master Protocol Pack Modbus®/C++ Editions;
- **Modbus Poll**: simulatore shareware di un master ModBus per il test e il debug;
- **"MODBUS Serial RTU + TCP/IP Simulator" di Conrad Braam**: simulatore freeware scritto in Visual C++ 6.0, che offre possibilità di testing della maggior parte delle funzioni ModBus.

Tutti gli applicativi utilizzati sono consigliati da ModBus-IDA, un'organizzazione non-profit indipendente il cui scopo è diffondere il protocollo della Modicon affinché questo si riesca ad affermare quale standard *de facto* in ambito industriale.
Per il testing e la calibrazione del protocollo TCP/IP lwIP è stato utilizzato il software freeware **Ethereal**, che consente di monitorare il traffico su reti ethernet, di stillare statistiche sull'efficienza della stessa ed è inoltre un ottimo strumento per il testing dei protocolli over ethernet.
Per la verifica delle due versioni del protocollo AmicLan sono stati utilizzati due simulatori di master AmicLan già presenti in azienda:

- **Heron 4**: simulatore prodotto da Amic S.r.l. che consente l'interrogazione di uno slave AmicLan su tutte le query previste dal protocollo e supporta la comunicazione su seriale e su TCP/IP;

- **Footograph 1.8**: simulatore di un master che pilota una fotocamera, che consente l'esecuzione di *Ping* e la richiesta dello scatto di una foto.

3.4.8 Le scelte d'integrazione

Per poter integrare i moduli sviluppati sarà necessario apportare qualche modifica al codice attualmente sviluppato, così da eliminare le incompatibilità e da ottenere codice il più ottimizzato possibile.
Come prima cosa si è reso necessario rendere compatibili i prototipi di interfaccia del protocollo AmicLan per le due versioni *over TCP/IP* e *over USB*. Per fare ciò, si propone un'interfaccia unica contenente entrambe le tipologie di parametri formali, la cui forma è la seguente:

```
int AmicLan_protocollo(struct tcp_pcb * pcb,
                       struct DataConnection * ws,
                       unsigned char[] aml_PDU_req,
                       unsigned int num_byte_req,
                       unsigned char[] aml_PDU_resp)
```

La scelta del protocollo da utilizzare sarà guidata dalle seguenti specifiche:

- Se i puntatori di tipo *tcp_pcb* e *DataConnection* sono NULL, allora il protocollo chiamante è l'USB e i parametri validi sono *aml_PDU_req*, *num_byte_req* e *aml_PDU_resp*;
- Se i puntatori di tipo *tcp_pcb* e *DataConnection* sono diversi da NULL, allora il protocollo chiamante è il TCP/IP e i parametri *aml_PDU_req*, *num_byte_req* e *aml_PDU_resp* vengono trascurati;

L'ultima modifica da apportare riguarda la gestione delle interruzioni: dal momento che sia l'interfaccia seriale che quella USB vengono

accedute in polling, è necessario che nessuna si blocchi in attesa di un dato, come invece succede nell'implementazione del ModBus sviluppata. Andrà quindi modificato il protocollo ModBus inserendo un flag che verrà settato dalla routine di interrupt dell'interfaccia seriale in seguito alla lettura del valore del timer, al fine di segnalare la terminazione della ricezione di un PDU; ancora, si dovrà modificare anche la funzione *Mb_read* in modo che non sia bloccante, spostando i controlli sullo stato dei timer e della trasmissione nella routine suddetta.

4. Conclusioni

Il progetto ELIK, sviluppato nel contesto di questa tesi, ha messo in luce nuove tematiche relative alla domotica. Lo stato dell'arte del settore, infatti, propone architetture che permettono l'integrazione di impianti domestici tradizionali, senza però tenere conto delle specificità che possono caratterizzare i singoli ambienti della casa. Con ELIK, invece, si è cercato di colmare questa lacuna con l'aggiunta di un ulteriore livello di dettaglio nella struttura di un impianto domotico, dotando così l'ambiente cucina di un sottosistema automatizzato ed aperto in grado di comunicare con i sistemi domotici esistenti.

La flessibilità del sistema, garantita dall'utilizzo di tecnologie di ultima generazione, consente un alto grado di espandibilità che potrà portare, in sviluppi futuri, all'ampliamento della gamma dei potenziali servizi offerti in funzione delle esigenze dell'utente e delle richieste del mercato; inoltre, pone la piattaforma sviluppata come base per molteplici tipologie di applicazioni e scenari, che consentiranno il miglioramento della qualità della vita in cucina.

Al fine di ottimizzare ulteriormente l'architettura realizzata, potrà essere eseguita un'ulteriore indagine per affiancare al protocollo proprietario AmicLan ulteriori protocolli aperti di livello applicazione: ad esempio, sul protocollo TCP/IP si potrebbe implementare il protocollo ModBus over TCP/IP. La possibilità di dotare il sistema di una infrastruttura di comunicazione aperta a livello firmware avrebbe l'effetto di una maggiore appetibilità sul mercato dei fornitori di elettrodomestici, che potrebbero adattare l'architettura ad eventuali soluzioni custom.

Rispetto alle soluzioni esistenti, quindi, ELIK offre, oltre alle caratteristiche per cui è stato ideato, una maggiore adattabilità alle

diverse necessità dell'utente. In merito a ciò, poiché è impensabile lo sviluppo di applicazioni personalizzate, si potrebbero sviluppare diverse categorie di prodotti che, pur basate sulla stessa infrastruttura, offrano servizi differenti.

Volendo estendere ancora le precedenti considerazioni, il sistema può essere visto come un'infrastruttura per lo sviluppo di ulteriori architetture distribuite, non necessariamente legate al vano cucina o all'automazione domestica stessa. Un'architettura siffatta potrebbe, infatti, essere utilizzata anche in ambito industriale, ad esempio per il controllo di forni industriali, o anche per la supervisione dell'intero impianto mediante l'interazione con eventuali dispositivi di sicurezza preesistenti, se compatibili. La scelta del protocollo ModBus, in questo caso, offre grandi potenzialità anche se, per rendere ancora più integrabile l'intera architettura, si potrebbe pensare, sempre nel contesto di sviluppi futuri, di implementare ulteriori protocolli industriali come il ProfiBus, il FieldBus, l'Etercat e così via.

Concludendo, il progetto ELIK si pone non solo come sistema per l'automazione dell'ambiente domestico cucina, ma anche come infrastruttura di base per la risoluzione di problematiche che richiedano l'utilizzo di architetture distribuite. Questa caratteristica lascia quindi aperta alla fantasia del committente la scelta dei possibili sviluppi futuri e offre all'azienda una soluzione evoluta adattabile a contesti diversi.

Appendice A1. Descrizione delle funzioni del protocollo ModBus

Funzione	Descrizione
Mb_test_crc	Verifica il CRC di un pacchetto.
Mb_calcul_crc	Calcola il CRC di un pacchetto e lo accoda allo stesso.
Mb_Init	Inizializza la porta seriale e il timer necessari alle elaborazioni ModBus.
SetBits	Imposta il valore del bit all'indirizzo passato come parametro al valore desiderato.
GetBits	Estrae il valore del bit all'indirizzo passato come parametro.
sendException	Crea il pacchetto di eccezione del tipo passato come parametro.
ReadHReg	Controlla che la richiesta di lettura di *holding register* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
ReadIReg	Controlla che la richiesta di lettura di *input register* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
WriteReg	Controlla che la richiesta di scrittura di un *holding register* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
WriteMulReg	Controlla che la richiesta di scrittura di più *holding register* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
WriteCoil	Controlla che la richiesta di scrittura di un *coil* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
ReadCoils	Controlla che la richiesta di lettura di *coil* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
WriteMulCoil	Controlla che la richiesta di scrittura di più *coil* sia corretta

	e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
ReadDInputs	Controlla che la richiesta di lettura di *discrete input* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
RWMulReg	Controlla che la richiesta di lettura e scrittura di più *holding register* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
ReadExStatus	Controlla che la richiesta di lettura del byte di stato delle eccezioni sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
Diagnostic	Controlla che la richiesta di *Return Query Data* sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
SlaveID	Controlla che la richiesta di lettura delle informazioni sullo slave sia corretta e, se così, crea il pacchetto di risposta appropriato, altrimenti crea un pacchetto di eccezione.
Mbs_read	Effettua polling sulla periferica UART finché non si riceve un PDU ModBus corretto.
Mbs_write	Scrive sulla periferica UART un PDU ModBus.
Mb_DLslave	Esegue le procedure di protocollo di livello Data Link Layer ModBus.
Mb_ALslave	Esegue le procedure di protocollo di livello Application Layer ModBus.

Appendice A2. Descrizione delle funzioni del protocollo lwIP

Funzioni del memory management

Funzione	Descrizione
pbuf_init	Crea un pool di PBUF_POOL_SIZE pbuf di tipo PBUF POOL, puntato dal puntatore *pbuf_pool*.
pbuf_alloc	In base al layer da cui arriva la chiamata, viene calcolato lo spazio necessario per lo/gli header. Successivamente si alloca (in maniera distinta a seconda del tipo di memoria utilizzata) la quantità di memoria necessaria per contenere sia il pbuf, che lo/gli header e si inizializzano le eventuali strutture dati. La funzione restituisce il pbuf allocato o il primo pbuf se si tratta di una catena di pbuf.
pbuf_realloc	Riduce una catena di pbuf alla lunghezza desiderata. A seconda della lunghezza desiderata, i primi pbuf in una catena potrebbero essere saltati e lasciati invariati. Gli ultimi pbuf nuovi nella catena saranno ridimensionati e i restanti liberati.
pbuf_header	Sistema il puntatore al payload per nascondere o rivelare gli headers nel payload. Ai PBUF_ROM e ai PBUF_REF non è possibile modificare la dimensione, quindi la chiamata fallisce. Viene fatto un controllo affinché l'incremento della dimensione dell'header non sposti il puntatore del payload nella parte anteriore del buffer. Restituisce zero in caso di successo, non zero in caso di fallimento.
pbuf_ref	Incrementa il contatore riferimento del pbuf.
pbuf_free	Dereferenzia una catena di pbuf o una coda e dealloca tutti i pbuf non più utilizzati nella testa di questa catena o coda. Decrementa il contatore di riferimenti dei pbuf: se raggiunge zero, il pbuf viene deallocato. Per una catena di pbuf questo si ripete per ogni pbuf nella catena, fino al primo pbuf che ha un contatore di riferimenti non nullo dopo il decremento. Quindi, se tutti i contatori di riferimenti sono ad uno, la catena viene liberata. Restituisce il numero di pbuf che vengono eliminati dalla catena.

pbuf_clen	Conta il numero di pbuf nella catena passata come parametro.
pbuf_cat	Concatena due pbuf (ognuna può essere una catena di pbuf).
pbuf_chain	Concatena due pbuf (o catene di pbuf) insieme. La chiamata non libera la memoria del secondo pbuf, quindi è necessario chiamare la pbuf_free(t) una volta che si è smesso di utilizzarlo.
pbuf_take	Crea un PBUF_POOL (o PBUF_RAM) come copia di un PBUF_REF. Usato per accodare pacchetti per conto dello stack lwIP, come una coda "ARP based". Penetra nella catena di pbuf e rimpiazza ogni buffer PBUF_REF con PBUF_POOL (o PBUF_RAM), e ognuno prende una copia del dato riferito.
pbuf_dechain	Estrae il primo pbuf da una catena di pbuf. Restituisce ciò che rimane della catena.
pbuf_queue	Aggiunge un pacchetto alla fine di una coda.
pbuf_dequeue	Rimuove un pacchetto dalla testa di una coda.

Funzioni dell'interfaccia di rete

Funzione	Descrizione
netif_init	Inizializza la lista delle interfacce a NULL.
netif_add	Aggiunge un'interfaccia di rete alla lista delle interfacce. I parametri sono: - *netif*: una struttura di tipo *netif* preallocata; - *ipaddr*: indirizzo IP per la nuova interfaccia; - *netmask*: maschera di rete per la nuova interfaccia; - *gw*: indirizzo IP del gateway di default per la nuova interfaccia; - *state*: dato "opaque" passato alla nuova interfaccia; - *init*: funzione di callback che initializza l'interfaccia; - *input*: funzione di callback chiamato per passare I pacchetti di ingresso allo stack; Restituisce l'interfaccia, o NULL se fallisce.
netif_set_addr	Imposta gli indirizzi dell'interfaccia con i valori passati come parametri.

netif_remove	Elimina l'interfaccia di rete passata come parametro dalla lista delle interfacce.
netif_find	Restituisce un'interfaccia di rete dato il suo nome. Il nome passato come parametro deve essere nella forma "et0", dove le prime due lettere sono il campo *name* nella struttura netif e il numero è il campo *num*.
netif_set_default	Imposta l'interfaccia passata come parametro come interfaccia di default.
netif_set_up	Attiva un'interfaccia, abilitando ogni elaborazione del traffico.
netif_set_down	Disattiva un'interfaccia, disabilitando ogni elaborazione del traffico.
netif_is_up	Controlla se un interfaccia è attiva.

Funzioni del layer IP

Funzioni	**Descrizione**
ip_route	vd. Paragrafo 3.4.5.3
ip_input	vd. Paragrafo 3.4.5.3
ip_output	vd. Paragrafo 3.4.5.3
ip_output_if	vd. Paragrafo 3.4.5.3
icmp_input	Gestisce i pacchetti ICMP ECHO rispondendo con un ECHO REPLY.
icmp_dest_unreach	Richiamata dallo strato di trasporto, invia un ICMP DESTINATION UNREACHABLE.
icmp_time_exceeded	Richiamata dalla funzione *ip_forward()*, invia un pacchetto ICMP TIME EXCEEDED.

Funzioni del layer UDP

Funzioni	**Descrizione**

udp_new	Alloca un nuovo PCB, inizializza i suoi campi a zero e imposta il campo TTL. Restituisce il PCB creato.
udp_remove	Rimuove dalla coda il PCB passato come parametro.
udp_bind	Collega un indirizzo IP e una porta locali ad un PCB e lo inserisce nella lista dei PCB attivi. Se la porta non viene specificata viene scelta una porta non utilizzata. Restituisce un codice d'errore se esiste già un PCB collegato a questa coppia.
udp_connect	Associa un indirizzo IP e una porta remoti ad un PCB. Se il PCB non è nella lista dei PCB attivi lo inserisce.
udp_disconnect	Imposta a zero l'indirizzo IP e la porta remota.
udp_recv	Associa la funzione *recv* passata come parametro al PCB passato come parametro.
udp_sendto	Invia dati a un indirizzo specificato utilizzando UDP. I parametri sono: - *pcb*: PCB usato per inviare i dati; - *pbuf*: catena di pbuf dei pbuf da inviare; - *dst_ip*: indirizzo IP di destinazione; - *dst_port*: porta UDP di destinazione. Se il PCB aveva già un'associazione di indirizzi remoti, questi verranno ripristinati dopo l'invio dei dati.
udp_send	Invia dati usando UDP. I parametric richiesti sono: - *pcb*: PCB usato per inviare i dati; - *pbuf*: catena di pbuf contenente i datagrammi da inviare;
udp_input	Elabora un datagramma UDP ricevuto in ingresso. Dato un datagramma UDP (come catena di pbuf) questa funzione trova un corrispondente PCB e lo demultiplexa su questo. - *pbuf*: pbuf da demultiplexare su un PCB; - *netif*: interfaccia di rete dalla quale si è ricevuto il datagramma.
Udp_init	Inizializza le code di PCB a NULL.

Funzioni del layer TCP

Funzioni	**Descrizione**
tcp_init	Inizializza lo strato TCP.

tcp_tmr	Routine di interruzione che gestisce il timer: richiama periodicamente le funzioni che gestiscono i timer interni del TCP.
tcp_close	Chiude una connessione mantenuta dal PCB passato come argomento.
tcp_abort	Abortisce una connessione inviando un segmento di RST all'host remoto e cancella il blocco di controllo locale. Questo viene fatto quando una connessione viene chiusa per carenza di memoria.
tcp_bind	Collega la connessione a un numero di porta e a un indirizzo IP locali. Se l'indirizzo IP non viene specificato (il parametro è NULL) viene usato l'indirizzo IP dell'interfaccia di uscita.
tcp_listen	Imposta lo stato della connessione a LISTEN: essa sarà quindi in grado di accettare connessioni in ingresso. Il PCB è riallocato utilizzando la versione ridotta della struttura dati, di modo da consumare meno memoria. Settare una connessione a LISTEN è un processo irreversibile.
tcp_recved	Questa funzione viene richiamata dall'applicazione quando ha elaborato i dati; l'obiettivo è quello di annunciare una finestra più larga quando i dati sono stati processati.
tcp_new_port	Individua un numero di porta non utilizzato.
tcp_connect	Connette ad un altro host. La funzione connected non è utile nel caso in cui non si utilizzino le API.
tcp_slowtmr	Viene chiamata ogni 500 ms e implementa il timer di ritrasmissione e il timer che rimuove i PCB che sono stati nello stato TIME-WAIT per un tempo eccessivo. Inoltre incrementa diversi timer, come il timer di inattività associato ad ogni PCB.
tcp_fasttmr	Viene richiamata ogni 250 ms si occupa della trasmissione di delayed ACK.
tcp_seg_free / tcp_segs_free	Dealloca un segmento / una lista di segmenti TCP.
tcp_setprio	Imposta la priorità di una connessione.
tcp_seg_copy	Restituisce una copia del segmento TCP passato come argomento.
tcp_kill_prio	Chiude tutte le connessioni attive che hanno priorità

	minore di *prio*.
tcp_kill_timewait	Chiude, se esiste, la connessione che è rimasta per più tempo nello stato TIME-WAIT.
tcp_alloc	Alloca lo spazio per un nuovo PCB: se non ci sono PCB liberi prima prova a "chiudere" quello che è rimasto per più tempo nello stato TIME-WAIT, e se esso non esiste "chiude" tutti quelli con priorità minore del parametro *prio*. Se anche quest'operazione fallisce restituisce NULL.
tcp_new	Crea un nuovo PCB senza collocarlo in alcuna lista.
tcp_arg	Usata per specificare l'argomento da passare alla funzione di callback.
tcp_accept	Usata per specificare la funzione da richiamare quando è stata instaurata una connessione di tipo LISTEN con un altro host.
tcp_sent	Usata per specificare la funzione da chiamare quando un dato TCP è stato consegnato con successo all'host remoto.
tcp_recv	Usata per specificare la funzione da eseguire quando una connessione TCP riceve dati.
tcp_poll	Usata per specificare la funzione che deve essere chiamata periodicamente dal TCP. L'intervallo è specificato in termini di intervalli di *coarse timer interval*, ossia 500 millisecondi.
tcp_pcb_purge	Rimuove ogni dato bufferizzato e libera la memoria del buffer.
tcp_pcb_remove	Come sopra, con la differenza che questa rimuove anche il PCB. Prima della rimozione invia tutti i delayed ACK.
tcp_next_iss	Calcola il numero di sequenza iniziale per le nuove connessioni.
tcp_input	L'elaborazione d'ingresso iniziale del TCO. Questa verifica l'header TCP, demultiplexa il segmento fra i PCB e lo passa alla funzione *tcp_process()* che implementa la macchina a stati finiti del TCP. Questa funzione viene chiamata dallo strato IP (nella *ip_input()*).
tcp_listen_input	Chiamata dalla *tcp_input()* quando arriva un segmento da una connessione nello stato LISTEN. In

	questo stato, controlla segmenti di tipo SYN, crea un nuovo PCB se possibile, e risponde con un SYN+ACK.
Tcp_timewait_input	Chiamata dalla *tcp_input()* quando arriva un segmento da una connessione nello stato TIME – WAIT.
tcp_process	Chiamata dalla funzione *tcp_input()*, implementa la macchina a stati finite del TCP. In alcuni stati viene chiamata la funzione *tcp_receive()* per ricevere dati. L'argomento *tcp_seg* verrà liberato dal chiamante (*tcp_input()*) non appena il puntatore *recv_data* nel PCB è settato.
tcp_receive	Chiamata dalla *tcp_process()*, controlla se un dato segmento è un ACK per dati in uscita, e, in caso affermativo, libera la memoria dei dati bufferizzati. Quindi il segmento viene collocato in una delle code di ricezione (*recved* o *ooseq*). Se il segmento viene bufferizzato, il pbuf viene riferito da pbuf_ref così che non venga liberato finché non è stato rimosso dal buffer. Se il segmento in ingresso costituisce un ACK per un segmento che è stato usato per la stima del RTT, questo viene stimato qui.
tcp_parseopt	Fa il parsing delle opzioni contenute nell'header del segmento in ingresso.
tcp_send_ctrl	Accoda un dato con i seguenti parametri: no data, no length, *flags*, copy=1, no optdata, no optdatalen.
tcp_write	Scrive un dato da inviare, ma non viene inviato immediatamente. Essa attende nell'aspettativa che più dati vengano inviati presto (poiché esso verrebbe inviato più efficientemente se combinato con altri). Per spingere il sistema ad inviare subito il dato è sufficiente chiamare la *tcp_output()* dopo aver chiamato questa.
tcp_enqueue	Accoda o dati o opzioni TCP (ma non entrambi) per la trasmissione. I parametri sono: - *pcb*: PCB della connessione TCP sulla quale vogliamo accodare il dato; - *arg*: puntatore al dato da accodare per l'invio;. - *len*: lunghezza del dato in byte; - *flags*: flag del segmento da inviare; - *copy*: uno se il dato deve essere copiato, zero se il dato è non volatile e può essere riferito;

	- *optdata*; - *optlen*;
tcp_output	Controlla cosa può essere inviato e lo invia.
Tcp_output_segment	Invia effettivamente un segmento TCP su IP.
tcp_rst	Invia un segmento di reset (RST flag).
tcp_rexmit_rto	Riaccoda tutti i segmenti unacked per la ritrasmissione.
tcp_rexmit	Sposta il primo segmento unacked nella coda dei segmenti non inviati.
tcp_keepalive	Invia un segmento di KEEPALIVE, usato nel controllo di flusso per verificare che la finestra non si sia riaperta.

Appendice A3. Descrizione delle funzioni del protocollo USB

Funzioni	Descrizione
USBInit	Inizializza l'hardware USB e imposta lo stack USB installando le funzioni di callback di default.
USBGetDescriptor	Scorre la lista dei descrittori USB installati e cerca di trovare il descrittore USB specificato.
USBSetConfiguration	Configura la periferica in funzione dell'indice di configurazione specificato e dell'impostazione alternativa scorrendo la lista dei descrittori USB installati.
HandleStdDeviceReq	Funzione locale per gestire una *standard device request*.
HandleStdInterfaceReq	Funzione locale per gestire una *standard interface request*.
HandleStdEndPointReq	Funzione locale per gestire una *standard endpoint request*.
USBHandleStandardRequest	Funzione locale per gestire le *standard request*.
USBRegisterCustomReqHandler	Registra una funzione di callback per le *custom device request*.
USBHwCmd	Funzione locale per inviare un comando all'*USB protocol engine*.
USBHwCmdWrite	Funzione locale per inviare un comando e dati all'*USB protocol engine*.
USBHwCmdRead	Funzione locale per inviare un comando all'*USB protocol engine* e leggere dati.
USBHwEPRealize	Realizza un endpoint, ossia gli riserva dello spazio di buffer; prima di poter usare un endpoint è necessario eseguire questa procedura.

USBHwEPEnable	Abilita o disabilita un endpoint.
USBHwEPConfig	Configura un endpoint e lo abilita.
USBHwRegisterEPIntHandler	Registra una funzione di callback per gli eventi di endpoint.
USBHwRegisterDevIntHandler	Registra una funzione di callback per lo stato della periferica.
USBHwRegisterFrameHandler	Registra una funzione di callback per il frame.
USBHwSetAddress	Imposta l'indirizzo USB.
USBHwConnect	Connette o disconnette dal bus USB.
USBHwNakIntEnable	Abilita le interruzioni per la condizione NAK. Per gli endpoint di ingresso si genera un NAK quando l'host vuole leggere un dato dalla periferica, ma nel buffer di endpoint non è presente niente. Per gli endpoint di uscita si genera un NAK quando l'host vuole scrivere un dato nella periferica, ma il buffer di endpoint è pieno.
USBHwEPIsStalled	Estrae la proprietà *stalled* di un endpoint.
USBHwEPStall	Imposta la proprietà *stalled* di un endpoint.
USBHwEPWrite	Scrive un dato in un buffer di endpoint.
USBHwEPRead	Legge un dato da un buffer di endpoint.
USBHwConfigDevice	Imposta lo stato *configured*.
USBHwInit	Inizializza l'hardware USB.
_HandleRequest	Funzione locale per gestire una richiesta chiamando uno dei gestori di richieste installati.
StallControlPipe	Funzione locale per porre in "stallo" l'endpoint di controllo.
DataIn	Invia il successivo "tronco" di dati all'host.
USBHandleControlTransfer	Gestisce trasferimenti di dati in ingresso

	e uscita nell'endpoint zero.
USBRegisterRequestHandler	Registra una funzione di callback per gestire le richieste.
BulkOut	Funzione locale per gestire i dati di tipo *bulk* in ingresso.
BulkIn	Funzione locale per gestire i dati di tipo *bulk* in uscita.
HandleClassRequest	Funzione locale per gestire le richieste di classe USB-CDC.

Appendice B. Riferimenti

Mailing list LPC2000
http://tech.groups.yahoo.com/group/lpc2000/

Manuale del GNU project Debugger
http://sourceware.org/gdb/download/onlinedocs/

Sito web della iSystem
http://www.isystem.com

Sito web della Keil
http://www.keil.com

Sito web della Modbus Organization, Inc.
http://www.modbus-IDA.org

Sito web della NXP (Philips)
http://www.standardics.nxp.com/

Sito web dell'ARM Corporation
http://www.arm.com/

Sito web dell'IEEE ***IEEEXplore release 2.3***
http:///ieeexplore.ieee.org

Sito web del progetto GNU
http://www.gnu.org

Bibliografia

AA. VV. *74HC/HCT573 Data Sheet*, Philips Semiconductors, 1990

AA. VV. *DP83848C PHYTER® Data Sheet*, <http://www.national.com>, National Semiconductor Corporation, 2007

AA. VV. *ARM7TDMI-S Technical Reference Manual*, <http://www.arm.com>, 1999

AA. VV. *Domotica, il libro bianco*, <http://www.laboratoriodomotica.it/>, 2005

AA.VV. *Errata Sheet LPC2378*, <http://www.nxp.com>, ver. 1.1, 2007

AA. VV. *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE Inc., 2001

AA. VV. *LM2596 SIMPLE SWITCHER® Data Sheet*, <http:// www.national.com>, National Semiconductor Corporation, 2002

AA. VV. *uA78M00 SERIES Positive-Voltage Regulators Data Sheet*, Texas Instrument Inc., 2005

AA.VV. *LPC2148 User manual*, <http://www.nxp.com>, rev. 01, 2005

AA.VV. *LPC2364/66/68/78 User manual*, <http://www.nxp.com>, rev. 01, 2007

AA.VV. *M25P10 Data Sheet*, ST Microelectronics, rev. 2.6, 2002

AA.VV. *MAX200–MAX211/MAX213 Data Sheet*, Maxim, rev. 6, 2003

AA.VV. *ModBus Application Protocol Specification*, <http://www.Modbus-IDA.org>, ver. 1.1a, 2004

AA.VV. *MODBUS over Serial Line - Specification and Implementation Guide*, <http://www.Modbus-IDA.org>, ver. 1.01, 2006

AA.VV. *Protel 99 SE – Designer's Handbook,* Protel International Limited, 2000

AA.VV. *SN75HVD10 Data Sheet*, Texas Instrument Inc., 2003

AA.VV. *Universal Serial Bus Specification*, <http://www.usb.org>, rev. 2.0, 2000

AA.VV. *Universal Serial Bus Class Definitions for Communication Devices*, <http://www.usb.org>, ver. 1.1, 1999

M. Campanai, M. Griffi, C. Paggetti, F. Tarchi, M. Traversi *Edifici intelligenti: opportunità per le imprese e per gli utilizzatori*, <http://www.firenzetecnologia.it/domotica>, 2003

A. Dunkels *Design and Implementation of the lwIP TCP/IP Stack*, Swedish Institute of Computer Science, 2001

S. Perini *Protocollo AmicLan versione 2.0*, Amic s.r.l, rev. 1.6, 2004

M. Ferri, M. Perrone, M. Traversi *Lo stato dell'arte della domotica: scenari futuribili e promesse non mantenute*, <http://www.firenzetecnologia.it >, 2006

T. Martin *The Insider's Guide To The Philips ARM7-Based Microcontrollers*, Hitex (UK) Ltd., 2005

L. L. Peterson, B. S. Davie *Reti di calcolatori*, Milano, Apogeo, 2004